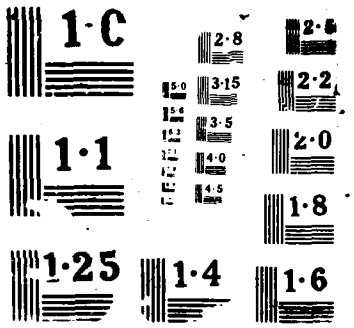
AD-A189 749 A UNIX-BASED INTERACTIVE VHDL (HARDWARE DESCRIPTION LANGUAGE) SIMULATOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. H H KODAMA 1/1

UNCLASSIFIED 07 DEC 87 AFIT/GE/ENG/87D-33 F/G 12/5 NL


1·0
2·8
2·5
5·0
3·15
2·2
5·6
3·5
6·3
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

(1)

A UNIX-BASED INTERACTIVE VHDL SIMULATOR

THESIS

Harvey H. Kodama
Second Lieutenant, USAF

AFIT/GE/ENG/87D-33

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GE/ENG/87D-33

A UNIX-BASED INTERACTIVE VHDL SIMULATOR

THESIS

Harvey H. Kodama
Second Lieutenant, USAF

AFIT/GE/ENG/87D-33

AFIT/GE/ENG/87D-33

A UNIX-BASED INTERACTIVE VHDL SIMULATOR

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of Master of Science in Engineering

Harvey H. Kodama

Second Lieutenant, USAF

December 1987

Preface

This thesis involved the design of a VHDL simulator in the C programming language to be operated in a UNIX environment. The simulator was developed concurrently with the VHDL analyzer. Together, the VHDL analyzer and simulator make a significant contribution to the AFIT VHDL Environment (AVE), and provide the academic community with a VHDL toolset that operates under the UNIX operating system.

The development of the VHDL simulator provided an opportunity to study the area of software engineering as it applies to VLSI design. I wish to express my appreciation to those VHDL group members whose input helped to make the thesis effort the success it was. In particular, I wish to acknowledge my thesis advisor, Maj Joe DeGroat, for his guidance and support throughout the thesis effort. I would also like to thank Capt Randy Bratton, who I've turned to on many occasions with my "C" questions, and CPT Mike Dukes, who helped navigate me through the VMS operating system, where the commercial VHDL simulator resides.

Finally, I would like to thank my wife, Mary, for her encouragement and understanding throughout the AFIT experience, enduring the hardship of separation and expressing genuine concern in my thesis progression.

Table of Contents

List of Figures

List of Tables

Abstract

This thesis effort investigated and implemented a UNIX-based VHDL subset interactive simulator. The simulator was written in the C programming language. The simulator is capable of handling many of the VHDL operators, sequential assignment statements, and concurrent statements. Although the simulator is a subset, the data structures were designed to incorporate the features of the complete VHDL version 7.2.

A UNIX-based Interactive VHDL Simulator

1. Introduction

1.1. Background

The Very-High-Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL), developed for the Department of Defense (DoD) by the Air Force Wright Aeronautical Laboratories (AFWAL), is an extremely important vehicle for the development and insertion of VHSIC technology in the 1990's (Dewey and Gadient, 86). Based on the programming language Ada, VHDL consists of structural building blocks used to design complex VLSI systems. Some of the features of VHDL include the ability to define memoried and combinatorial elements, create user-defined types, and describe design hierarchies. Although other hardware description languages exist, none match VHDL's total capabilities when working with VLSI and VHSIC circuits (Waxman, 86). The AFIT VHDL Environment (AVE) (Carter *et.al.*, 87) is an advanced prototype UNIX-based VHDL programming and design environment that was begun in 1986. The VHDL Simulator, a prototype kernel of which was developed by MAJ William Lynch (Lynch, 86), is a critical link in the AVE.

1.2. Problem Statement

Universities are unable to investigate the capabilities of VHDL due to a lack of acceptable software tools (analyzers, simulators, etc.). Since many universities use the UNIX operating system, a VHDL tool set that operates under UNIX is necessary if VHDL is to be utilized by the academic community.

This project is to develop MAJ Lynch's VHDL Simulator kernel into a version 7.2 VHDL subset Simulator. This VHDL Simulator, in conjunction with the UNIX-based VHDL Analyzer

(Bratton, 87) provides the academic community with the necessary tools to investigate the capabilities of VHDL. The simulator is written in the C programming language (Kernighan and Ritchie, 78), capable of running on VAX-11/785 and ELXSI 6400 computers under the AT&T System V and 4.2 and 4.3 bsd UNIX operating systems. Input is in VHDL Intermediate Access (VIA) format, produced by Capt Randolph Bratton's VHDL Analyzer. Interface with the analyzer is critical, since the simulator requires specific information from the VHDL description. The simulator's operation was validated, analyzed, and simulation speed was examined.

1.3. Hardware Description Languages

1.3.1. Definition and Requirements

Formally, an HDL is defined as a language for describing, documenting, simulating and synthesizing digital systems with the aid of a computer (Su, 77). The origin of hardware description languages (HDLs) to describe digital system designs date back to 1939, with Shannon's work on switching circuits (Chu, 74). Interest in HDLs has grown, and a myriad of languages has been developed to describe computer systems. HDLs allow the designer to express asynchronous operations, parallel control, and the architecture of a hardware system in a precise, yet concise description (Barbacci, 85).

An HDL must be able to express the sequence of actions and structure of the components needed to implement those actions -- e.g., flip-flops, memory units, ALUs, and bus structures. It must be able to describe all of these, whatever its complexity (Su, 77).

1.3.2. Need for HDLs

Due to the rapid advancement of semiconductor device fabrication technology, the performance and speed realized from an integrated circuit has increased substantially. As circuit density increased, design details could no longer be conveyed in circuit diagrams (Lipovski, 77). Although the type number and pin connections were represented, circuit diagrams did not reveal essential design details such as internal functions being performed, indications of legal and illegal connections, and principles of operation (Lipovski, 77). The overwhelming complexity and density of circuit designs necessitated the use of design teams in the development of VLSI technology. The design team concept, however, was accompanied with the problem of communicating design information concisely, accurately, and efficiently (Lynch, 86).

In order to alleviate communication problems, circuit designers incorporated the use of HDLs into the design process. The need for HDLs was recognized by Yaohan Chu in 1974. When asked, "Why do we need computer hardware description languages?" Chu offered the following reasons (Chu, 74):

- they serve as a means of communication among design engineers;
- they permit precise yet concise descriptions;
- they provide documentation;
- they are amenable to simulation on a computer;
- they aid greatly in computer aided design (CAD).

By utilizing HDLs, hardware design engineers have a means of communicating their designs in a precise, self-documenting language. These features contribute to hardware design reusability and maintainability in digital systems. When HDLs are simulated on a computer and utilized in a CAD environment, both design time and design costs are reduced substantially.

A hardware simulator permits its designer to simulate, usually before construction, hardware designs (Lynch, 86). The designer can create a software model for a hardware system, exercise the model with a set of input stimuli, and observe the output. Logic simulation is especially valuable for VHSIC designs since design errors are very costly and breadboarding is impractical (d'Abreu, 85).

1.3.3. Problems with HDLs

The recognition of the ability of HDLs has led to the proliferation of weak, ineffectual languages. When a simulator was written, the designer felt entitled to write his own HDL; he felt that it would be easier to design a new HDL than to adapt existing ones to his application (Lipovski, 77). In describing the state of HDLs, Lipovski says, "We seem to be confounded in our own Tower of Babel -- everybody is talking a different language but nobody is communicating." (Lipovski, 77). In a survey of HDLs, Su concluded that although many languages exist, none have been widely utilized to design, describe, or document digital systems (Su, 74).

The problem has been that although HDLs existed for specific applications, none were satisfactory over the range required for a large hardware design project (Lynch, 86). In 1977, Stephen Su suggested the use of a common HDL for circuit designers (Barbacci, 85). In 1981, the Department of Defense acknowledged the need for an HDL for Very-High-Speed Integrated Circuits (VHSIC) class designs (Shahdad, 85). The DoD hoped to establish the VHSIC Hardware Description Language (VHDL) as the standard language for VHSIC designs.

1.4. VHSIC Hardware Description Language (VHDL)

1.4.1. Background

In the 1970s, the DoD initiated the development of the Ada programming language in reaction to the crisis in software development. Requirements for the Ada programming language were that it incorporate software engineering concepts such as structured programming, information hiding, and data abstraction (Booch, 87).

When analyzing the problem of how to effectively communicate information on VHSIC designs consisting of over 250,000 transistors, the DoD found the basic constructs implemented in Ada were consistent with the requirements for a new HDL (Lynch, 86).

1.4.2. Development and Definition of VHDL

In 1981, the Institute for Defense Analyses (IDA) was tasked with the requirements analysis for the VHSIC Hardware Description Language (Aylor, 86). In the process of developing VHDL, existing HDLs and their environments were analyzed, extracting the major advantages of each; this insured that no existing HDL capabilities were overlooked in developing VHDL (Aylor, 86). In 1982, the VHDL program organization was formulated; in 1983, the VHDL program was launched when the team of Intermetrics, IBM, and Texas Instruments was awarded the contract to design and implement the VHDL support environment software (Dewey, 86; Gilman, 86). Intermetrics, the prime contractor for the program, was tasked with designing the language and establishing the support environment architecture (Gilman, 86).

1.4.3. Goals and Attributes of VHDL

The specific goals of VHDL are to reduce circuit design time and insert VHSIC technology into military systems (Dewey, 86). More generally, VHDL strives to meet the HDL goals noted earlier (precise communication, documentation, simulation, and CAD applications). The primary goal of the DoD is to establish VHDL as a standard design automation interface tool and documentation agent (Dewey, 86). This would greatly simplify complex system design and facilitate VHSIC insertion into electronic systems.

1.4.4. VHDL in a VMS Environment

The VHDL tool set developed by Intermetrics was written in Ada, to be operated in a VAX-VMS environment. The Air Force Institute of Technology (AFIT) was selected as a site for beta-testing the VMS-based VHDL tools. The VHDL Analyzer and Simulator were ported onto AFIT's VMS VAX-11/785 (Carter, 87). The installation procedure was complicated due to inadequate installation instructions and some file protection problems. Furthermore, it was later discovered that the software contained version dependencies that interfered with the proper operation of the VHDL tools when later releases were installed (Carter, 87).

1.4.5. AFIT VHDL Environment (AVE)

In recognition of the deficiencies of the Ada/VMS-based VHDL tool set implementation, the AFIT VHDL environment (AVE) was established in 1986. The AVE is a prototype UNIX-based VHDL programming and design environment in which UNIX-based software tools are developed in the C programming language. The key attribute of software developed in C is its portability; that is, the capability of installing it on different UNIX-based computer systems with little or no modifications to the software. In 1986 AFIT thesis efforts, a prototype VHDL Analyzer (Frauenfelder, 86) and a prototype VHDL Simulator (Lynch, 86) were developed. The validation benchmarks of these prototype versions offer proof of the UNIX-based design concept (Frauenfelder, 86; Lynch, 86). A UNIX-based environment is favored because it is the operating system utilized by most universities, and because of the inherent portability of C software.

To facilitate the insertion of VHSIC technology into university programs and government systems, the prototype Analyzer and Simulator had to be developed into VHDL CAD tools.

1.5. Assumptions

Completion of this thesis assumes a mapping of source code to a VHDL intermediate format. A working analyzer is not essential, as the intermediate files could be synthesized using a text editor if necessary.

1.6. Materials

- VHDL Users' Guide and Reference Manual (Intermetrics, 85a, Intermetrics, 85c)
- C Reference (Kernighan and Ritchie, 78)
- A UNIX-based computer with a C compiler; (An ELXSI 6400 super-minicomputer and Sun-3 workstation was used to meet this requirement.)

1.7. Approach

The overall design philosophy in this research effort was to incrementally implement language features of VHDL into the simulator. Since the signal assignment was considered the most fundamental behavioral statement, it was the first to be implemented. As features of the language were implemented, tested, and validated for the Analyzer and Simulator, Beta versions were released. Successive Beta versions incorporated additional language features, always maintaining end-to-end throughput from VHDL source to simulation output.

In the next chapter, the system design philosophy is discussed; the features of the simulator are introduced, and the design approach and methodology are described, and the notion of a pre-compiled simulator is justified. Chapter 3 is devoted to a discussion of the steps involved in creating compilable C code from the intermediate form, and an explanation of data structures used for simulation. The chapter is concluded by conceptually exercising a simulation for a VHDL description, showing the intermediate results in the simulation process. In chapter 4, the simulator is analyzed by listing the features of VHDL that were implemented in the simulator, the execution times for various phases in the simulation, size of files involved, and a comparison of simulation reports obtained from the VMS version and the UNIX version. Finally, in chapter 5, recommendations for future research in the area of VHDL simulation is discussed.

2. System Design

For the subset simulator to be deemed complete, it must meet performance requirements defined and described in this chapter. The performance requirements state the subset of the language the simulator must be capable of simulating, the simulator's user interface, its operating environment, and execution time and file size requirements.

2.1. Objectives

The primary objective of this research effort was to develop a subset VHDL simulator in the C programming language capable of running under UNIX; additionally, the simulator should be fast, have modest disk space requirements, and be easy to use. The simulator must read an intermediate form of VHDL and an optional input vector file.

2.2. VHDL Subset Simulator

For the UNIX-based simulator to be considered complete, it must meet the following criteria:

- It should correctly simulate the *signal assignment, enable, disable, assertion, if,* and *case* sequential statements; and the *process, concurrent assertion, conditional signal assignment,* and *selected signal assignment* concurrent statements, as described in the Language Reference Manual (Intermetrics, 85a).
- The simulator must have a user interface that includes an interactive mode which allow the user to set simulation breakpoints and view signal values or change signal values at breakpoints. The simulator must allow the user to specify an input stimuli file, allowing the user to provide input vectors for signals.
- It should complete the simulation, from intermediate form to output report generation, in less CPU time than its VMS-based counterpart. The disk space required for simulation under UNIX must also be less than that required by the VMS version.

Obtaining objective comparisons for execution time may be difficult due to differences in CPUs on available VMS and UNIX machines. Disk space, however, is absolute, and can be easily compared. From a software engineering perspective, the simulator must adhere to the principles of structured programming (modularity, module independence, etc.) and proper code documentation (Pressman, 87).

These software engineering principles are fully supported in the design of the simulator. The overall simulator is divided into two packages, the Build package and the Simulate package, each performing a distinct operation. Each of these packages is implemented by decomposition into hierarchically designed functions. Each function is documented with a header which describes what the function does, and explains the purpose of the parameters passed and local variables.

2.3. VHDL Intermediate Form

VHDL Intermediate Access (VIA), produced by Capt Bratton's VHDL Analyzer (Bratton, 87), serves as the intermediate form for the simulator. VIA provides the complete design description in the form of a syntax tree. Simulation from VIA is more manageable since the VHDL source has been syntactically and semantically verified, and has been reorganized into a tree structure that is easily parsed by the simulator. Additionally, many of the control structures (if-then-else, case, loop) are easily mapped from the syntax tree format of VIA to C code.

2.4. Interactive Capability

The simulator must provide interactive features that allow the user to provide input stimuli, set breakpoints, view signal values, change signal values, and terminate the simulation. Each of these features are described below.

2.4.1. Input Vector File

The simulator must provide the user with the ability to change the values of input stimuli. The input vector file permits the user to specify signals that are to be set, the time they are to be affected, and the value the signal will assume. For example, the user should be able to specify that signal A is set to '0' at time 0ns, '1' at time 5ns, '0' at time 15ns, etc. The input vector file is optional, and the user can specify any number of vectors in the file.

2.4.2. Breakpoints

The user should be able to set breakpoints in the simulation. When a breakpoint is specified, the simulation proceeds until the specified time is reached. At that time simulation is suspended, allowing the user the option of viewing intermediate values, altering values, or terminating the simulation.

2.4.3. View Signal Values

This function permits the user to view intermediate signal values during the course of a simulation. Following a breakpoint selection, the user can specify which signal values should be displayed. The signal values will be displayed to the console, and the user can display more signal values, continue the simulation, or terminate the simulation.

2.4.4. Change Signal Values

This function permits the user to alter the course of a simulation by forcing changes on one or more signal values by specifying a signal name followed by the value it is to assume. The user can repeat this function for as many signals as desired.

2.4.5. Terminate Simulation

This feature allows the user to abort a simulation without further processing.

2.5. Implementation Requirements

The simulator is implemented in the C programming language, to be operated under the AT&T System V and UNIX bsd 4.3 operating system on an ELXSI 6400, DEC VAX 11/785, and a SUN workstation.

2.6. Performance Requirements

Since the primary objective of the VHDL simulator is to simulate VHSIC class circuits, its memory requirements and execution speed were given particular attention when designing the data structures. Figure 2-1 shows the difference between the VHSIC class design and the less complex designs solved by Engineering Analysis Simulation Systems (Intermetrics, 84). The larger number of gates being simulated in VHSIC class designs (up to 100K) usually require vast amounts of memory and have very long execution times on large minicomputers and mainframe computers. The vast memory requirements and long execution times stifle productivity and especially impact users in a timesharing environment.

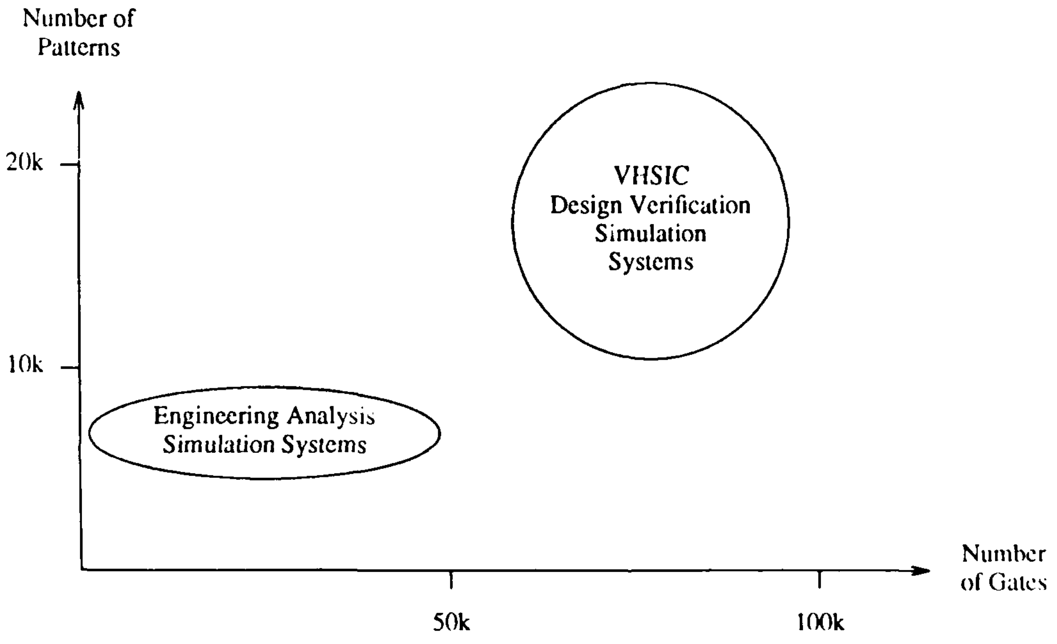


Figure 2-1. Complexity of VHSIC Class Designs.

The event-driven methodology used in this simulator is designed to help alleviate the vast memory requirements and long execution times inherent in VHSIC class simulation. Figure 2-2 shows the difference in efficiency between interpretive and precompiled simulators. Interpretive simulators are typically more efficient to set up, but less efficient as the number of test patterns increases into the VHSIC realm of designs (Intermetrics, 84).

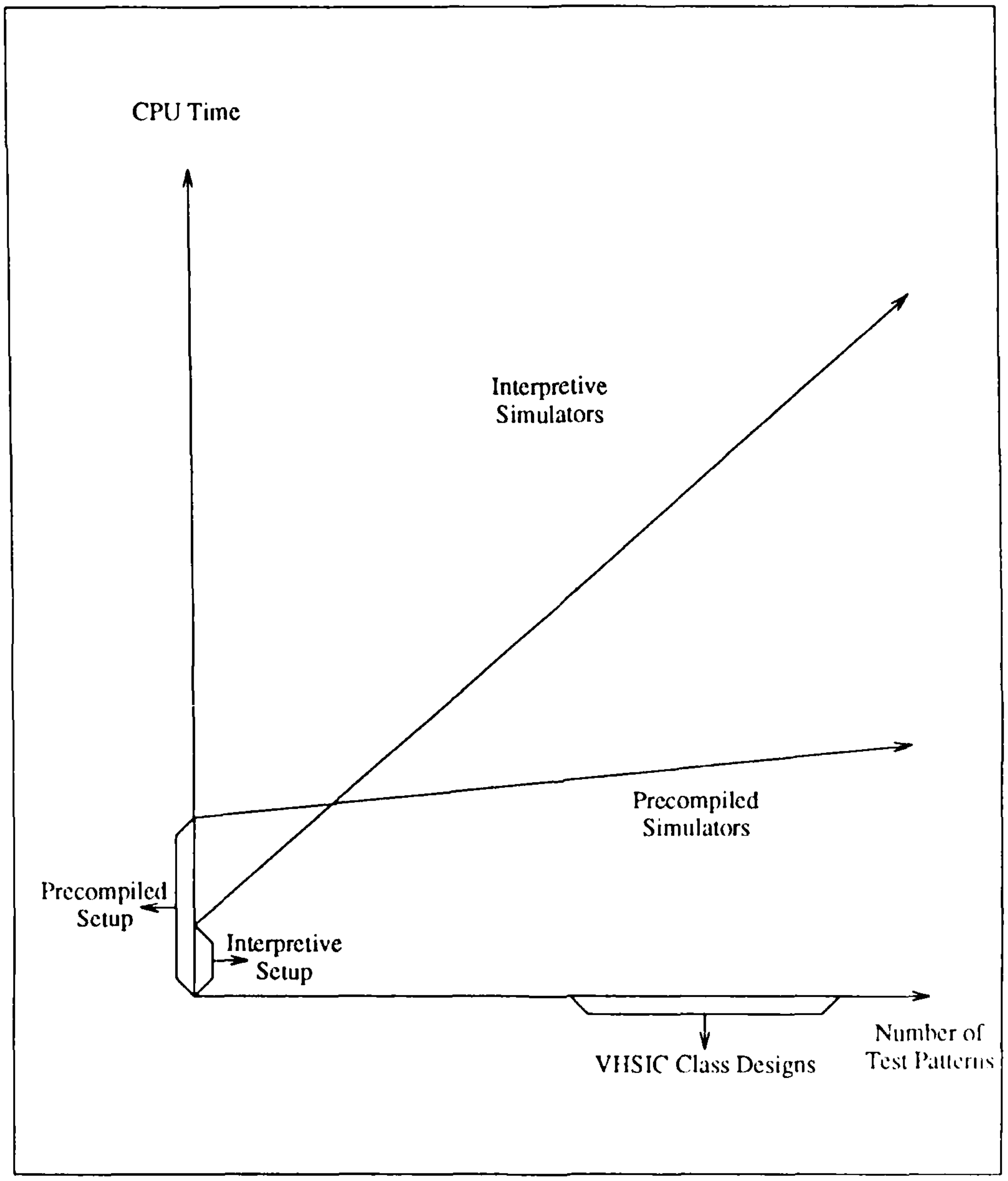


Figure 2-2. Precompiled vs. Interpretive Simulator Comparison.

2.7. Simulation Methodology

2.7.1. Approach

The two classes of simulators are compiler-driven and table-driven event-directed (or table-driven). Earlier simulators were compiler-driven, while the more modern ones are event-driven (d'Abreu, 85).

In digital circuits, usually only 10 to 15 percent of the circuit is active at any point in time (d'Abreu, 85). This makes it inefficient to simulate all elements. Instead, an event-driven simulator is based on changes in the state of signals. When a signal is scheduled to change value at some future time, both it's projected value and its scheduled time are maintained in linked structures called *event lists* and *time queues* (d'Abreu, 85).

This VHDL simulator is an event-driven simulator that maintains both a time queue and event lists. The data structures used to meet this requirement are described in detail in the next chapter.

2.7.2. Overall System Design

In the system design phase, an approach similar to the Structured Analysis and Design Technique (SADT) (Pressman, 87) was used to develop the software system. Using SADT-like diagrams, the system could be viewed from a very high level of abstraction, exposing data transformation and data flow. SADT-like diagrams were prepared for the overall system, the Build package and the Simulator package. At these high levels of abstraction, SADTs proved helpful in perceiving the interaction between the problem space and the solution space. At lower levels in the design, SADT-like diagrams were not as effective at describing and documenting modules; instead algorithmic descriptions were used.

The overall simulation execution flow is described below:

- Read VIA from file into internal structure (array).
- Parse the symbol table section of VIA, creating symbol table in memory.
- Parse the operation table section of VIA, starting from the highest level block statement. As the operation table is parsed, files containing executable C code are generated.

- The C modules just created are compiled and linked with the kernel of the simulator and the run-time library.
- The simulator is executed.
- Tailored reports can be generated.

The SADT-like diagram used to describe and document the simulator at the highest level of abstraction is depicted in Figure 2-3. The simulator obtains control information from the command line, via command line options, and from user responses, when operated in the interactive mode. The simulator obtains input from VIA, which describes the VHDL description, and from an optional vector file. Based on the inputs and control specifications, an output file that describes the behavior of the VHDL description is produced. If there were any errors in the input or in the user specified control, an appropriate message is displayed to the terminal.

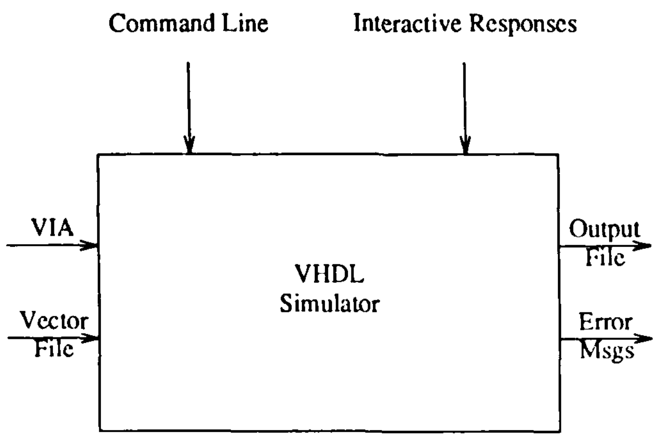


Figure 2-3. System Level SADT Diagram.

2.7.3. Build Package

The Build package reads VIA and creates the necessary C code for creating the static data structures and functions necessary for simulation. The SADT-like diagram used to design the Build package of the simulator is depicted in Figure 2-4. The user must specify the name of the VIA file on the command line. If the file does not exist, an appropriate message is displayed to the terminal, otherwise the specified VIA file is read, and C code that models the VHDL description is generated.

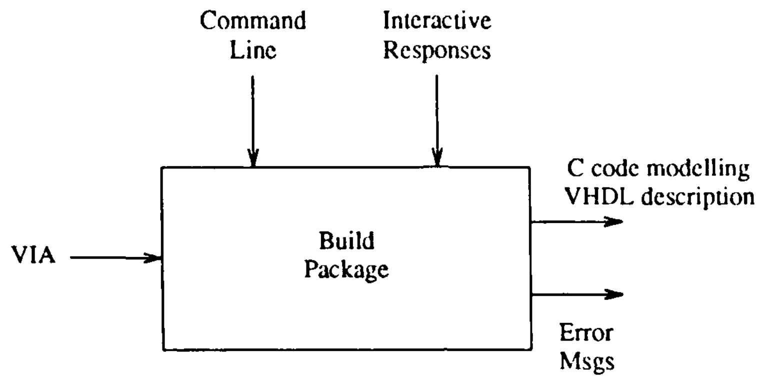


Figure 2-4. Build Package SADT Diagram.

The Build package creates the static data structures and the behavioral functions by performing the following:

- Read VIA file into an internal structure (array).
- Parse the symbol table section of VIA for all signals and variables declared.
- Parse the operation table section of VIA, starting from the highest level block statement. As the operation table is parsed, the C functions describing the description's behavior are incrementally generated.

In the next chapter, the sequence of steps described above are discussed in more detail, and illustrated by applying algorithm for the Build package to a VHDL description. The structure of the operation table is illustrated in the example, as well as the content of the C files generated.

2.7.4. Simulate Package

The Simulate package begins by compiling the C function(s) created in the Build package and linking them with the kernel of the simulator. All information from VIA is contained in the C functions, so the Simulate package does not access VIA.

The design of the Simulate package began by describing the data flow and control in a SADT-like diagram, which is depicted in Figure 2-5. When executing the Simulate package, the user may specify the name of an output file on the command line, otherwise a default name is used. During the course of a simulation, the user has the opportunity to set breakpoints, view intermediate results, or change signal values during the course of the simulation. When the Simulate package begins execution, the C code that models the VHDL description is compiled and linked with the kernel of the simulator. The simulator is executed, and an output file is created. If there were any errors in the simulation, the user will be notified by appropriate messages to the terminal.

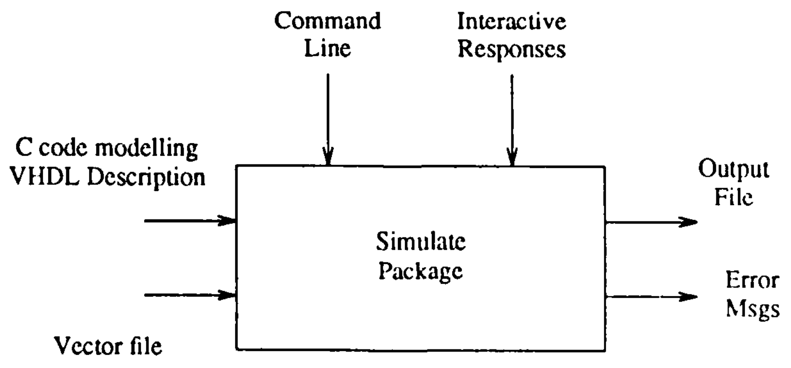


Figure 2-5. Simulate Package SADT Diagram.

The Simulate package is then executed, exercising the following cycle adapted from the VHDL tutorial:

1. If the next vector time is equal to the current simulation time, the value of each signal in the input vector is read sequentially. If the value of a signal in the input vector is different from the current value of the signal, the signal value is updated instantaneously and all processes sensitive to that vector are marked for execution. If the value of the signal in the input vector is the same as the current value of the signal, nothing happens. Signal values in the input vector are read and evaluated until the input vector has been exhausted.

2. If the current simulation time is the same as the first record in the time queue, all transactions in the event list for this time are evaluated. If the scheduled value is different from the current value of the signal, all processes sensitive to that signal are marked for execution. If the scheduled signal value is the same as the current value of the signal, nothing happens. Transactions in the event list for the current time are evaluated until the event list is empty.

3. All marked processes in the process table are executed, possibly posting transactions to signals' projected output waveforms.

4. Time advances. The global simulation time advances until the user specified termination time is reached or the time queue is empty.

3. Detailed Design

In the previous chapter, the requirements for the simulator were described, followed by a system level view of the Build and Simulate packages' functionality. In this chapter, the design of the Build and Simulate packages will be discussed in more detail with examples where appropriate; unique implementations of algorithms and data structures are emphasized in this discussion.

3.1. Build Package

The philosophy of the Build package is to read the intermediate form once and create C code that models the behavior of the VHDL description, as well as declarations and assignments for the static data structures in the simulation. The C code created in the Build package can then be compiled and linked with a simulator kernel and runtime library for execution.

The Build package begins by reading the VIA file into an array. As mentioned earlier, VIA is maintained as series of records that can be conceptually linked, forming a binary tree consisting of *first* and *next* nodes. When viewed as an array of records, each record has a *first* and a *next* field, which are the indices of other records in the array. The records in the array can alternately be viewed as nodes of a tree, where the *first* and *next* fields are viewed as pointers to other nodes. For a thorough explanation of the VIA format and contents, the reader is encouraged to read Bratton's work on the VHDL analyzer (Bratton, 87).

The tree is recursively traversed and evaluated, starting from the highest level BLOCK statement. As nodes are encountered, they are evaluated, and the C code modeling the VHDL description is incrementally generated.

The philosophy of the Build package provides runtime efficiency because VIA is read from a file into an internal representation, which is traversed in one pass, building C code that models the behavior of the VHDL description represented in VIA. By creating C code, elements in the VHDL description (signals, drivers, variables, etc.) can be represented as discrete elements.

If the elements in the VHDL description were not represented as discrete records, they would have to be represented in a linked list of records, or as an array of records. Implementing the data structure as a linked list would increase search times, while an array implementation would not be memory efficient.

3.2. Build Package Example

In this section, the execution of the Build Package is examined for the example of a three input XOR signal assignment statement. First the VHDL description is presented, followed by the VIA file generated after running the analyzer. The VIA format is presented in its untouched form, a formatted form, and the equivalent operation table in a visual tree structure. Finally, the C code generated by the Build package, and a pictorial representation of the static data structures needed for the simulator are discussed.

3.2.1. VHDL Description

The VHDL description for the three input XOR signal assignment statement is merely a process statement containing a sequential signal assignment statement. The VHDL description used in this example is given below.

```
entity XOR3

  (X,Y:  in BIT;                 -- one-bit addends
   Z:  in BIT;                   -- carry in
   F:  out BIT)                  -- one-bit sum

is
end XOR3;

architecture DATA_FLOW_IMPL of XOR3 is

   BLOCK_1:  block

   begin
     process (X, Y, Z)
       begin
      F <= X xor Y xor Z after 41 ns;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

3.2.2. VIA Representation

Depicted below is the untouched VIA file output by the analyzer. The first four entries in VIA indicate how many design units, symbol table, operation table, and string table entries there are in the VHDL description. In this example, there is 1 string table entry, 32 symbol table entries, 19 operation table entries, and 3 design units.

```
1 32 19 3
0 __ANY__ 0 0 0 0 -1 -1 -1 -1 -1
1 STANDARD 3 4 0 1 -1 -1 -1 -1 -1
2 INTEGER 2 1 0 1 -2147483648 2147483647 2 -1 -1
3 CHARACTER 2 2 0 1 128 0 127 -1 -1
4 BIT 2 2 0 1 2 0 1 -1 -1
5 BOOLEAN 2 2 0 1 2 0 1 -1 -1
6 FALSE 1 3 0 1 5 0 -1 -1 7
7 TRUE 1 3 0 1 5 1 -1 -1 -1
8 SEVERITY_LEVEL 2 2 0 1 4 0 3 -1 -1
9 NOTE 1 3 0 1 8 0 -1 -1 10
10 WARNING 1 3 0 1 8 1 -1 -1 11
11 ERROR 1 3 0 1 8 2 -1 -1 12
12 FAILURE 1 3 0 1 8 3 -1 -1 -1
13 NS 1 3 0 1 14 1 -1 -1 15
14 TIME 2 5 0 1 0 2147483647 8 13 -1
15 US 1 3 0 1 14 1000 -1 -1 16
16 MS 1 3 0 1 14 1000000 -1 -1 17
17 S 1 3 0 1 14 1000000000 -1 -1 -1
18 0 1 3 0 1 4 0 -1 -1 19
19 1 1 3 0 1 4 1 -1 -1 -1
20 REAL 2 6 0 1 -2147483648 2147483647 20 -1 -1
21 NATURAL 2 1 0 1 0 2147483647 2 -1 -1
22 POSITIVE 2 1 0 1 1 2147483647 2 -1 -1
23 STRING 2 3 0 1 22 3 1 0 -1
24 BIT_VECTOR 2 3 0 1 21 4 0 0 -1
25 XOR3 3 1 0 25 29 -1 -1 -1 -1
26 Y 1 1 9 25 4 -1 -1 -1 27
27 X 1 1 9 25 4 -1 -1 -1 -1
28 Z 1 1 9 25 4 -1 -1 -1 26
29 F 1 1 17 25 4 -1 -1 -1 28
30 DATA_FLOW_IMPL 3 2 0 30 25 18 -1 -1 -1
31 BLOCK_1 1 4 0 31 18 -1 -1 -1 -1
0 0 1 -1 -1 -1 -1 -1
1 9 16 -1 5 7 -1 -1
2 9 17 -1 4 27 4 -1
3 9 17 -1 4 26 -1 -1
4 9 17 -1 4 28 3 -1
5 15 17 -1 -1 2 15 -1
6 9 19 -1 4 29 14 -1
7 9 19 -1 4 27 8 -1
8 9 19 -1 4 26 -1 -1
```

```
9 367 19 -1 4 7 10 -1
10 9 19 -1 4 28 -1 -1
11 367 19 -1 4 9 -1 -1
12 11 19 -1 2 -1 -1 41
13 291 19 -1 2 12 11 -1
14 13 19 -1 -1 13 -1 -1
15 4 19 -1 -1 6 -1 -1
16 345 20 -1 -1 5 -1 -1
17 12 22 -1 -1 1 16 -1
18 301 22 31 -1 17 -1 -1
```

3.2.3. Formatted VIA

The VIA format presented in the previous section is not particularly easy to read. Therefore, a utility called **fmtvia**, for *format VIA*, was developed with the analyzer software package (Bratton, 87). The **fmtvia** program reads a VIA file and produces a formatted VIA file with descriptive labels for each of the fields, and replaces numbered field descriptions with symbolic names. When the VIA file of the previous section was run through fmtvia, the following output was produced. The listing below contains the same information as the VIA file of the previous section, just in a more readable format.

```
AFIT VHDL VIA-to-Text Utility  Revision: 2.0
string size = 1; symbol table entries = 32
op table entries = 19; design units = 3
0 name=__ANY__ scope_ref=0 ANY_CLASS ANY_UNIT
1 name=STANDARD scope_ref=1 design_unit package
      package_declarations=-1
2 name=INTEGER scope_ref=1 type integer_range min=-2147483648
      max=2147483647 typeref=2
3 name=CHARACTER scope_ref=1 type enumeration #_of_const=128
      first=0 last=127
4 name=BIT scope_ref=1 type enumeration #_of_const=2 first=0 last=1
5 name=BOOLEAN scope_ref=1 type enumeration #_of_const=2 first=0
      last=1
6 name=FALSE scope_ref=1 object constant typeref=5 value=0
      initialize=-1 next=7
7 name=TRUE scope_ref=1 object constant typeref=5 value=1
      initialize=-1 next=-1
8 name=SEVERITY_LEVEL scope_ref=1 type enumeration #_of_const=4
      first=0 last=3
9 name=NOTE scope_ref=1 object constant typeref=8 value=0
      initialize=-1 next=10
10 name=WARNING scope_ref=1 object constant typeref=8 value=1
      initialize=-1 next=11
```

```
11 name=ERROR scope_ref=1 object constant typeref=8 value=2
     initialize=-1 next=12
12 name=FAILURE scope_ref=1 object constant typeref=8 value=3
     initialize=-1 next=-1
13 name=NS scope_ref=1 object constant typeref=14 value=1
     initialize=-1 next=15
14 name=TIME scope_ref=1 type physical #_of_units=8 min=0
     max=2147483647 baseref=13
15 name=US scope_ref=1 object constant typeref=14 value=1000
     initialize=-1 next=16
16 name=MS scope_ref=1 object constant typeref=14
     value=1000000 initialize=-1 next=17
17 name=S scope_ref=1 object constant typeref=14
     value=1000000000 initialize=-1 next=-1
18 name=0 scope_ref=1 object constant typeref=4 value=0
     initialize=-1 next=19
19 name=1 scope_ref=1 object constant typeref=4 value=1
     initialize=-1 next=-1
20 name=REAL scope_ref=1 type real_range min=-2147483648
     max=2147483647 typeref=20
21 name=NATURAL scope_ref=1 type integer_range min=0
     max=2147483647 typeref=2
22 name=POSITIVE scope_ref=1 type integer_range min=1
     max=2147483647 typeref=2
23 name=STRING scope_ref=1 type array indexref=22
     elementref=3 lo_bound=1 hi_bound=<>
24 name=BIT_VECTOR scope_ref=1 type array indexref=21
     elementref=4 lo_bound=0 hi_bound=<>
25 name=XOR3 scope_ref=25 design_unit interface port=29
     generic=-1 interface_declarations=-1 directives=-1
26 name=Y scope_ref=25 object signal typeref=4
     initialize=-1 next=27 other_info=( port mode=in )
27 name=X scope_ref=25 object signal typeref=4
     initialize=-1 next=-1 other_info=( port mode=in )
28 name=Z scope_ref=25 object signal typeref=4
     initialize=-1 next=26 other_info=( port mode=in )
29 name=F scope_ref=25 object signal typeref=4
     initialize=-1 next=28 other_info=( port mode=out )
30 name=DATA_FLOW_IMPL scope_ref=30 design_unit architecture
     interface=25 block=18
31 name=BLOCK_1 scope_ref=31 object label first_stmt=18 next=-1
0 NO_OP line=1 label=-1 type=-1 first=-1 next=-1
1 SYM_REF line=16 label=-1 type=5 symref=7(TRUE) next=-1
2 SYM_REF line=17 label=-1 type=4 symref=27(X) next=4
3 SYM_REF line=17 label=-1 type=4 symref=26(Y) next=-1
4 SYM_REF line=17 label=-1 type=4 symref=28(Z) next=3
5 SENSITIVITY_LIST line=17 label=-1 type=-1 first=2 next=15
6 SYM_REF line=19 label=-1 type=4 symref=29(F) next=14
7 SYM_REF line=19 label=-1 type=4 symref=27(X) next=8
8 SYM_REF line=19 label=-1 type=4 symref=26(Y) next=-1
9 XOR line=19 label=-1 type=4 first=7 next=10
10 SYM_REF line=19 label=-1 type=4 symref=28(Z) next=-1
11 XOR line=19 label=-1 type=4 first=9 next=-1
12 I_VALUE line=19 label=-1 type=2 value=41 next=-1
13 AFTER line=19 label=-1 type=2 first=12 next=11
```

```
14 WAVE line=19 label=-1 type=-1 first=13 next=-1
15 SIG_ASSIGN line=19 label=-1 type=-1 first=6 next=-1
16 PROCESS line=20 label=-1 type=-1 first=5 next=-1
17 GUARD line=22 label=-1 type=-1 first=1 next=16
18 BLOCK line=22 label=31 type=-1 first=17 next=-1
```

3.2.4. Operation Table Tree

The simulator reads the unformatted VIA from disk. After reading the header information, the simulator knows how many design units, symbol table entries, operation table entries, and string table entries to expect. The symbol table and operation table are read, line by line, into an array; the string table entries are read, character by character, into a string. Initially, the symbol table is used for signal and variable declaration. Once all signals and variables have been declared, the symbol table is used indirectly by the operation table when signals appear in the VHDL description. Figure 3-1 shows the operation table visualized as a tree. The operation table tree provides the same information found in the operation table section of VIA. Each node in the operation table is labeled with the node number and name of the operation table entry. Note the SYM_REF (symbol reference) nodes, which refer back to the symbol table.

For the three input XOR VHDL description, there is a single BLOCK statement (node 18), which consists of a single process statement (node 16). The process has a sensitivity list of the symbols X, Y, Z, represented by nodes 2, 3, and 4 respectively. Inside the process statement there is a signal assignment statement (SIG_ASSIGN, node 15), which assigns to symbol F (node 6; SYM_REF) the waveform described in the subtree with root at node 14 (WAVE). The AFTER node specifies after what time the signal F is to take on the expression pointed to by AFTER's *first* pointer, in this case, a three input XOR.

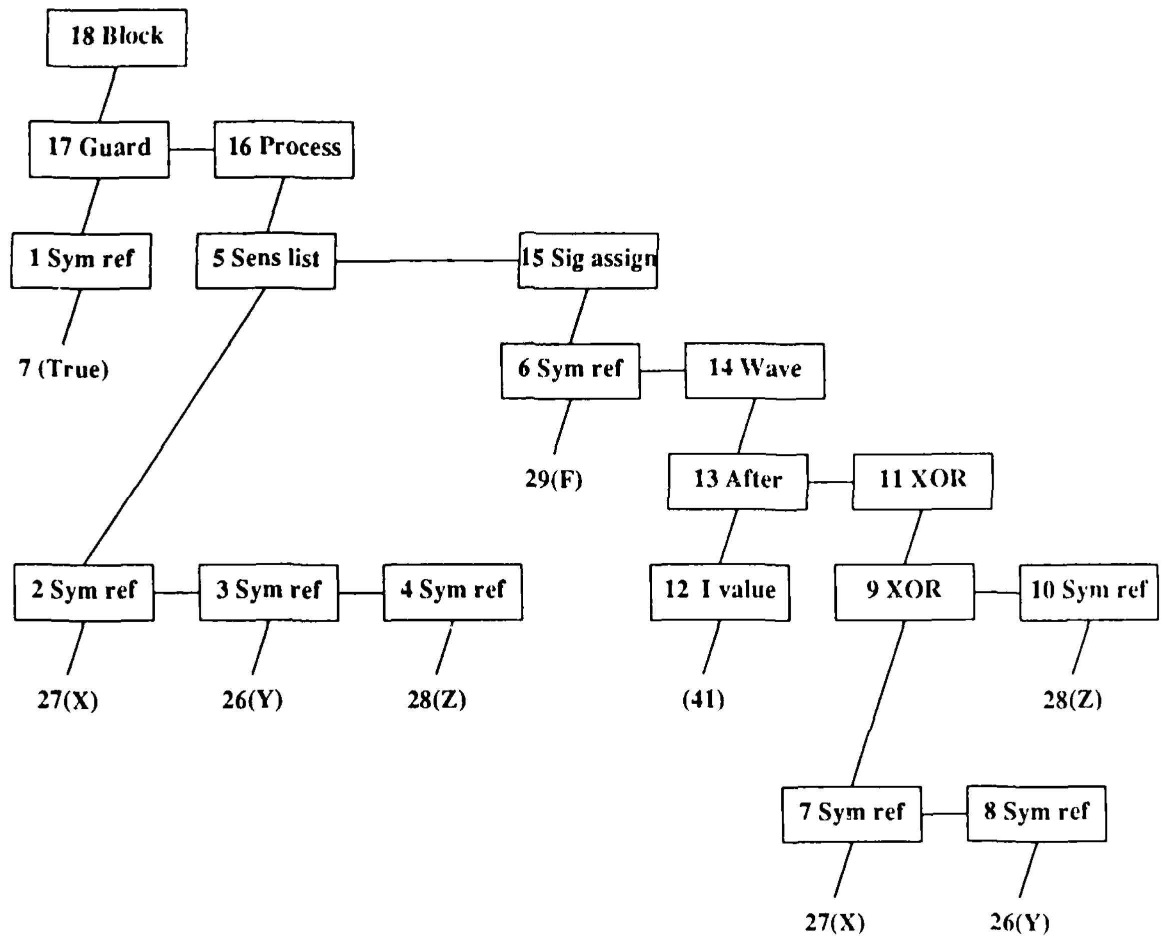


Figure 3-1. VIA Operation Table Structure

3.2.5. C Functions

The complete C code that is generated by from the above VHDL description and VIA is listed here.

```
#include <stdio.h>
#include "sim_stru.h"
int *s27_p1;
int *s28_p1;
int *s26_p1;
int s26_sens_proc[2];
int s27_sens_proc[2];
int s28_sens_proc[2];
int s29_sens_proc[1];
Driver *pd1[1];   int *ps1[3];

sim_initialize(sig_array, pt)
Signal *sig_array[];
Process_table_item pt[];
{
  Signal *Newsig();
  Signal *s26;
  Signal *s27;
  Signal *s28;
  Signal *s29;
  char *strsave();
  extern int s26_sens_proc[];
  extern int s27_sens_proc[];
  extern int s28_sens_proc[];
  extern int s29_sens_proc[];
  int i;
  Driver *Newdrv();
  int p16();
  extern Driver *pd1[]; extern int *ps1[];
  Driver *d29;
  /* end of declarations */

  /* create signals */
  s26 = Newsig(); s26->signame = strsave("Y");
  s27 = Newsig(); s27->signame = strsave("X");
  s28 = Newsig(); s28->signame = strsave("Z");
  s29 = Newsig(); s29->signame = strsave("F");
  sig_array[0] = s29;
  sig_array[1] = s27;
  sig_array[2] = s26;
  sig_array[3] = s28;
  /* declaring drivers */
  d29 = Newdrv(); d29->sig_ptr = s29;     s29->drv_ptr = d29;

  s26_sens_proc[0] = 0;  /* initially, no sensitive processes */
  s27_sens_proc[0] = 0;
  s28_sens_proc[0] = 0;
  s29_sens_proc[0] = 0;
  s26->sens_proc = s26_sens_proc;
  s27->sens_proc = s27_sens_proc;
  s28->sens_proc = s28_sens_proc;
  s29->sens_proc = s29_sens_proc;
  pt[1].proc_ptr = p16;
/* found a sensitivity list for a process */
  s27_sens_proc[++s27_sens_proc[0]] = 1;
      s27_pl = &s27_sens_proc[s27_sens_proc[0]];
  s28_sens_proc[++s28_sens_proc[0]] = 1;
      s28_pl = &s28_sens_proc[s28_sens_proc[0]];
  s26_sens_proc[++s26_sens_proc[0]] = 1;
      s26_pl = &s26_sens_proc[s26_sens_proc[0]];
  pdl[0] = d29;
  psl[0] = s27->cur_val;
  psl[1] = s26->cur_val;
  psl[2] = s28->cur_val;
/* process table filling of driver & signal list */
  pt[0].next = &pt[0];
  for(i=1; i<2; i++)
    pt[i].next = NULL;
  pt[1].dlist = pdl;    pt[1].slist = psl;
} /* end sim_initialize() */

/* VHDL Process(es) modelled in C code */
p16(dlist,slist)  /* process 16 */
Driver *dlist[];
int *slist[];
{
  Transact *Newtrans();
  extern TIME simtime;
  {  BOOLEAN transport = FALSE;
     Transact *newtrans1; /*wave*/
     newtrans1 = Newtrans();
     newtrans1->future_time = simtime + 41;
     *newtrans1->val = xor(xor(*slist[0], *slist[1]), *slist[2]);
     post_trans(dlist[0], newtrans1, transport);
  }
}
```

3.2.5.1. Static Data Structures

The C code begins by declaring all static data structures to be used in the simulation such as drivers, signals, and the process table entries. All these statements are declared inside a function called *sim_initialize*, which is invoked by the kernel of the simulator to set up the static data structures.

Each signal encountered in the symbol table is entered into an array, sorted by signal name. Each driver encountered in the operation table results in creating a driver structure that is doubly linked to the signal it affects, assuming each signal has only a single driver.

Each signal has a sensitive process list which holds indices into the process table for all processes sensitive to a signal. The sensitive process list is maintained as an array, with the first element of the array indicating how many processes the signal is sensitive to. In this example, no processes are sensitive to signal s29 ("F"), while s27 ("X"), s26 ("Y"), and s28 ("Z") each have one process sensitive to it, and its index in the process table is 1. (The zeroth position is not used to hold a process.)

3.2.5.2. Process Functions

In addition to the initialization routine, a C function is generated for each process. In this case, there is only one process, so there is only one C function. The function generated for the three input exclusive OR example is shown in above; the function is named p16 because it models a process which happens to be line 16 in the operation table. Names assigned to functions that model VHDL processes will always be P*n* where *n* is the line on which the process statement is found in the VIA file.

Each C function that models a process statement is passed two parameters: an array of pointers to drivers, and an array of pointers to signal values from the process table. When this process is executed, a new transaction is created, loaded with the appropriate time and value, and is posted to the projected output waveform of the appropriate driver.

3.3. Simulate Package

The Simulate package consists of three groups of data structures listed below.

- The driver and signal structures (static data structure)
- The process table (static data structure)
- The event lists and time queue (dynamic data structure)

The C code for the first two groups of data structures are generated in the Build package, and are compiled and linked with the kernel of the simulator. The third data structure is dynamically allocated and freed during the execution of the simulation.

3.3.1. Driver and Signal Structures

The driver structure is necessary to maintain, for each driver, the current value of the driver, a pointer to its projected output waveform, and a pointer to the signal that is affected by it. The fields of the driver structure are shown in Figure 3-2.

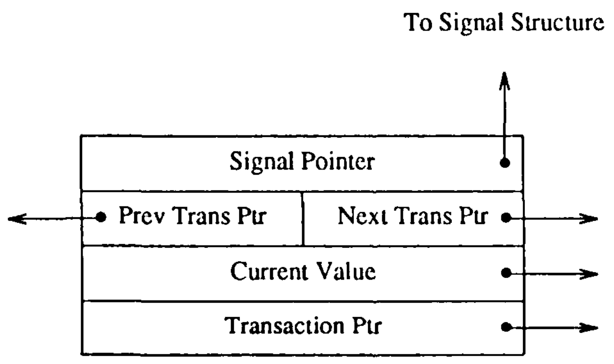


Figure 3-2. Driver Structure

In a VHDL description, the value of a signal is a function of the value of its drivers (Intermetrics, 84a). When a driver changes value, its associated signal must be re-evaluated. The first field depicted in the driver structure (signal pointer) is a pointer to the signal structure that is affected by a change in the driver value. The previous and next transaction pointers (Prev Trans Ptr and Next Trans Ptr) will be discussed later as part of the event list. The current value field on the driver structure is a pointer to the value of the driver's value. The value of a driver is not a fixed data size. A driver can have a value of type bit, character, integer, real, or bit vector. In order to conserve memory, all driver values are maintained as pointers to driver values. The final field is a transaction pointer which maintains a pointer to a driver's projected output waveform.

A signal structure is created for each signal in the symbol table. It maintains a pointer to an array of processes that are sensitive to changes in the signal value. Whenever the signal value changes, all processes sensitive to it are marked in the process table. The signal structure must also maintain a pointer to the current value of the signal, as well as a pointer to a string indicating the name of the signal. The driver list is a linked list of all drivers that the signal is a fuction of. The linked list of drivers is used by the bus resolution function to determine the value of a signal. The bus resolution function field in the signal structure is a pointer to a function that resolves the values of drivers a signal is sensitive to. In this simulator implementation, since only single drivers are allowed, the bus resolution function assigns the value of a driver to the signal it drives. Figure 3-3 depicts a single signal structure with its relationship to the driver structure and sensitive process list.

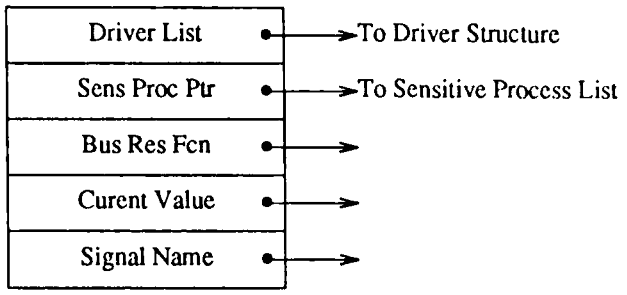


Figure 3-3. Signal Structure

3.3.2. Projected Output Waveform

Each driver in a simulation maintains a projected output waveform in which future time/value pairs are maintained. The simulator manages the projected output waveform as a linked list of times and pointers to values. As previously justified, the projected value of a driver is maintained as a pointer to accomodate the various data sizes in VHDL. A single element of a drivers' projected output waveform is depicted in Figure 3-4.

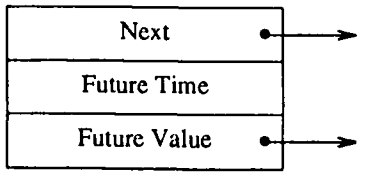


Figure 3-4. Projected Output Waveform Element

3.3.3. Process Table

The process table is set up during the Build phase, and consists of all processes in the VHDL description. A single process table entry is depicted in Figure 3-5. The process table is maintained as an array of records, each record consisting of the following fields: A pointer to a C function that simulates the process, an array of drivers that could be affected by the process firing, an array of signals that are needed to evaluate the process, and a next field used to maintain a list of all marked processes.

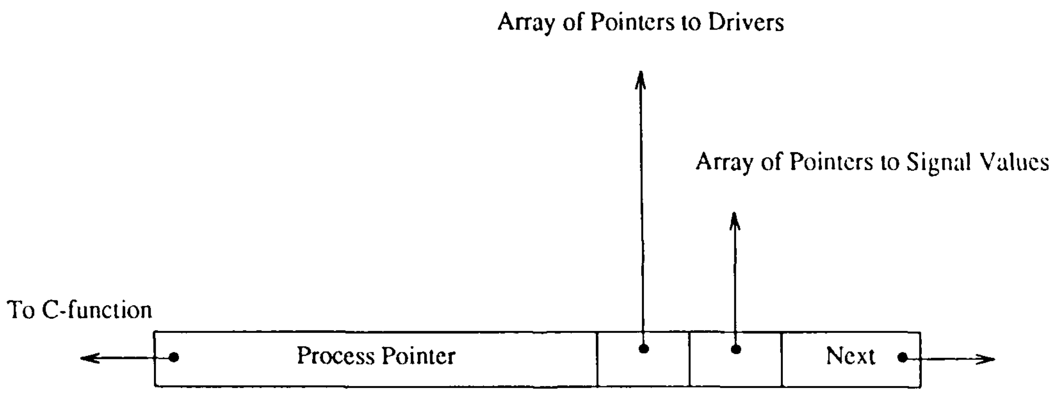

Array of Pointers to Drivers
Array of Pointers to Signal Values
To C-function
Process Pointer
Next

Figure 3-5. Process Table Entry

When the Build Phase of simulation is complete, the number of process is in the VHDL description is known, so the process table can be dimensioned as an array of exact size. Since the size of the process table is known and since it will never change, dynamic memory allocation (linked list) is not necessary. Each process table element maintains a pointer to a C function that models the VHDL process. The order of process execution is dynamic, making it difficult to execute functions by name since the function name would have to be imbedded in the C code.

By accessing the function by address, processes could be executed indirectly without explicitly providing the name of the C function.

When a process executes, both input parameters and output parameters must be specified. The input parameters are signal values that are needed to evaluate a process. They are provided by passing the process a list of pointers to signal values. When a process executes, drivers' projected output waveforms could be modified as a result of signal assignment statements in the process. By maintaining a linked list of all drivers affected by a process firing, the projected output waveform of each of these drivers can be accessed. By passing the addresses of signal values and projected output waveforms to a process, the simulation can be easily modified to handle component instantiation. When multiple instances of a component occur, a process table entry is created for each process, but references to the same process are resolved by having them all address the same C function, with each passing different signal values and projected output waveforms. The final field in a process table entry is a *next* field used for marking processes for execution. A linked list, initially null, contains all processes that have been marked for execution. When a signal value changes, all processes sensitive to that signal are marked by adding the *next* field to the linked list of marked processes. Before a process is marked for execution, its *next* field is examined. If it is null, the process is added to the linked list. If it's not null, then it's already part of the linked list, so nothing is done.

3.3.4. Event Lists and Time Queue

An event list is a doubly-linked structure that maintains all drivers whose first transaction on its projected output waveform is scheduled to occur at a given simulation time. For example, if three drivers had the first transaction on their respective projected output waveforms scheduled to occur 35ns into the simulation, each of these three drivers would be on the same event list. Figure 3-6 depicts this condition.

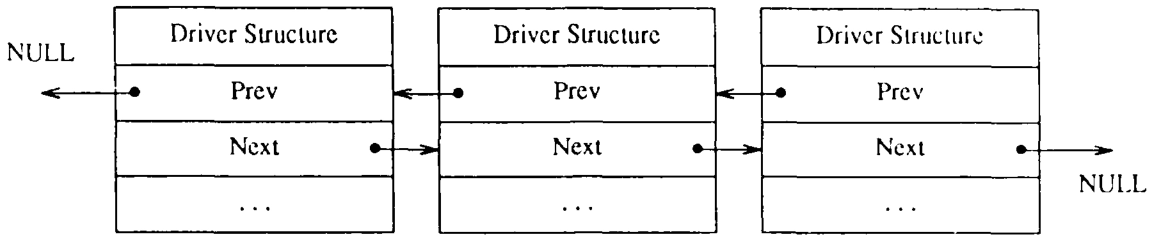


Figure 3-6. Event List for a Given Simulation Time

The time queue is dynamically allocated and freed during simulation. It is maintained as a singly linked list of records. Each record contains fields to hold a future simulation time, and head and tail pointers to point to the head and tail of the event list for the given simulation time. If there were an event list as described in the previous paragraph, then a time queue element would exist for 35ns as shown in Figure 3-7. Figure 3-8 depicts the general case of a time queue and its relationship to event lists.

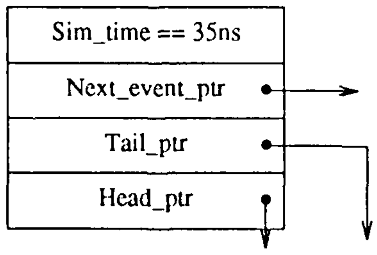


Figure 3-7. Time Queue

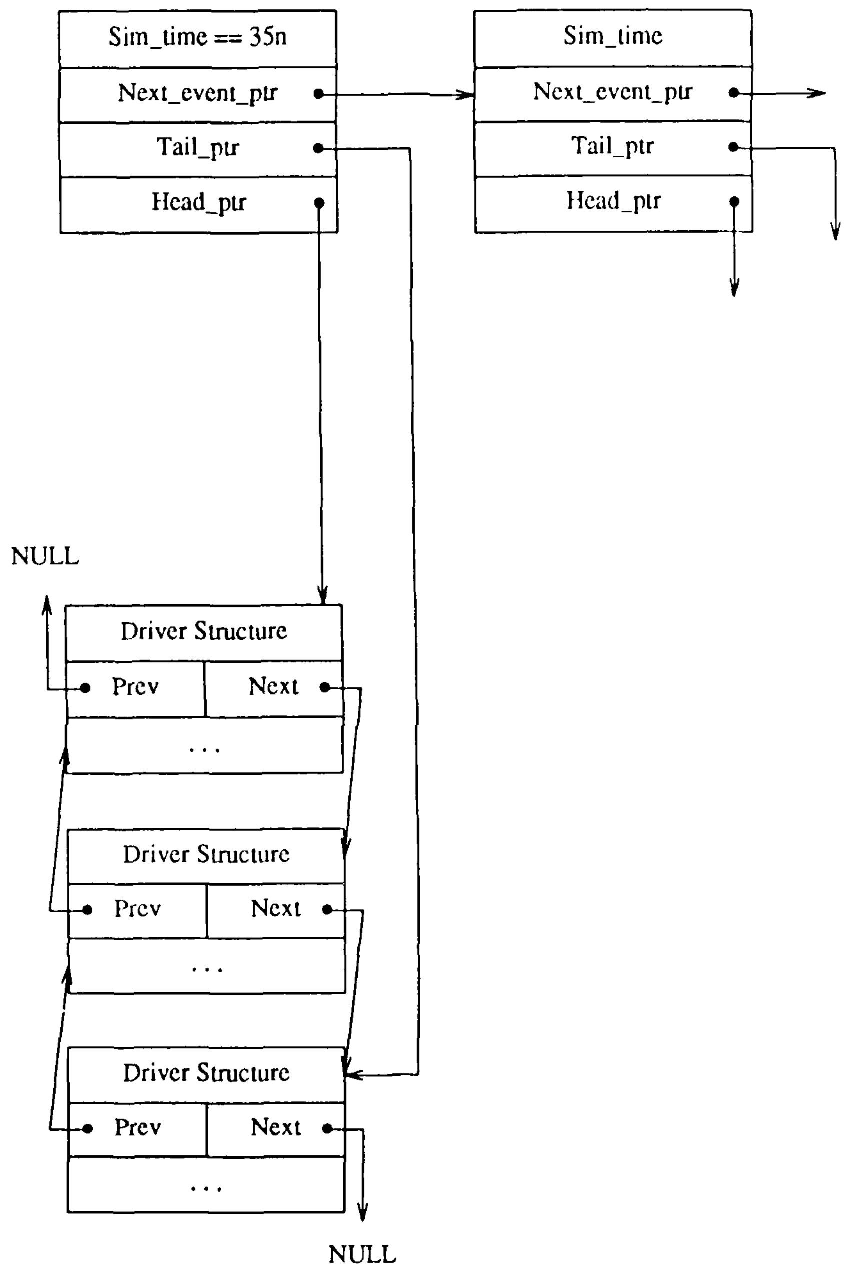


Figure 3-8. Time Queue and Event List

4. Results and Analysis

The UNIX-based VHDL simulator was incrementally designed, with the final product incorporating the major features of the language. This chapter presents the features incorporated into the simulator and analyzes the final product, comparing it to its VMS-based counterpart where appropriate. Both the UNIX-based and VMS-based simulators are evaluated for CPU time and disk storage requirements by simulating VHDL descriptions on each. The merit of the UNIX-based simulator is evaluated based on the language features implemented, its user's interface, and computer resource requirements.

4.1. Simulator Design

Since the analyzer and simulator were designed concurrently, the overall goal was to have end-to-end throughput; that is, to have the ability to analyze a VHDL description, produce the intermediate form (VIA), and simulate, producing an output report.

In order to obtain end-to-end throughput, the simulator was incrementally designed by taking a subset of VHDL, designing a prototype for it, and validating its operation. This was first done for sequential assignment statements. Both the analyzer and simulator were designed to handle sequential signal assignment statements. The results were validated, and the package was released as a Beta version. When the sequential assignment statement was validated, the simulator was modified to handle process statements, which could contain one or more sequential assignment statements.

Between the design capable of simulating sequential signal assignment statements and the design capable of simulating process statements, the basic algorithms did not change significantly, but the data structures required restructuring to accommodate the new features. Following the incorporation of process statements into the simulator, neither algorithms nor data structures required change, and the additional features were easily incorporated into the design, since the mapping of many of the control structures from VIA to C was largely one-to-one.

4.2. Simulator Operation

4.2.1. Command Line Options

Two of the requirements for the simulator are that it have a interactive user's interface, and that it be relatively easy to use. This section describes the interactive features implemented, which contribute to the simulator's ease of use.

When the user starts a simulation, information is provided via the command line and later from interactive responses from the user. On the command line, the user is permitted to specify any of the following options which perform the functions indicated. If an option is not selected, a default value is used.

-i simulate in interactive mode.

-o *out_file* designates the output transaction file.

-s *start_time* selects the simulation start time.

-t *term_time* selects the simulation termination time.

-v *vector_file* specifies the name of the input vector file.

By default, simulation is executed in non-interactive mode, the file *output.trn* is used for the output transaction file, 0 and 1ms are used for the simulation start and termination time respectively, and no input vector file is used if none is the -v option is not specified.

4.2.2. Interactive Features

If the -i option was specified on the command line, the user exercises the simulation in interactive mode. In the interactive mode, the user has the option of setting breakpoints, viewing signal values, changing signal values, setting and locking a signal to a value, unlocking a signal previously locked, running the simulation to the next breakpoint or until completion, running the simulation until the next time in the time queue, and terminating the simulation without further processing.

When the user starts the simulation in interactive mode, the data structures for simulation are initialized, and the user is prompted with the following menu:

Interactive Simulation Menu

? - Help

b - Set a breakpoint

v - View signal value(s)

c - Change signal value(s)

l - Set and lock a signal to a value

u - Unlock a previously locked signal

r - Run the simulation to next breakpoint or completion

n - Run the simulation until time advances

x - Terminate the simulation

The user is assisted through the simulation with sub-menus and is prompted for all required infoımation.

4.3. Program Results

4.3.1. Program Size

As described earlier, the UNIX-based VHDL Simulator consists of a Build package and a Simulate package. The Build Package consisted of 1,100 lines of C source source code, which produces an executable file 82,250 bytes in size. The Simulate package consist of 1600 lines of C source code, which produces an executable simulator with a minimum size of 90,700 bytes in size, depending on the size of the VHDL description.

4.3.2. Language Features Implemented

The simulator is capable of handling the predefined types **BIT**, **BOOLEAN**, **INTEGER**, and the physical type **TIME**, as well as enumeration types. The operators that are supported are listed below in Table 4-1.

Operators
logical operators: **and, or, nand, nor, xor, not**
relational operators: =, /=, <, <=, >, >=
arithmetic operators: +, -, *, /, **mod, rem**
miscellaneous operators: **, **abs**

Table 4-1. Operators Supported

The logical operators **and, or, nand, nor, xor,** and **not** are valid for the types **BIT** and **BOOLEAN.** All of the logical operators require two inputs (binary operators), with the exception of **not,** which is a unary operator. The standard relational operators are implemented. Each of the relational operators accepts two operands, which must be of the same type. The relational operation is performed, and returns a value of type **BOOLEAN.** The four standard arithmetic operators are also implemented for integer types, as well as the exponentiation operator and the absolute value operator.

Tables 4-2 and 4-3 list the sequential and concurrent statements that are implemented in the UNIX-based VHDL simulator, as described in the VHDL Language Reference Manual (Intermetrics, 85a).

Sequential Statements
• Signal Assignment Statement
• Enable/Disable Statements
• Assertion Statement
• If Statement
• Case Statement

Table 4-2. Sequential Statements Supported

Concurrent Statements
• Single Block Statement
• Process Statement
• Concurrent Assertion Statement
• Conditional Signal Assignment Statement
• Selected Signal Assignment Statement

Table 4-3. Concurrent Statements Supported

The operators and sequential and concurrent statements listed above have been tested and validated through analysis and simulation. The VHDL descriptions used to validate the operation of the simulator are included in the VHDL simulator test suite included in Appendix A.

4.3.3. Designs Tested

In this section, two circuits of practical interest are examined in detail, comparing the UNIX-based simulator's performance to the VMS-based simulator's performance. Factors such as time required to perform intermediate phases in a simulation, disk space requirements, and simulation results are examined.

4.3.3.1. Full Adder Description

4.3.3.1.1. UNIX-based Simulator

The VHDL description for the full adder evaluated is presented in Appendix B. The VHDL description was analyzed with the UNIX-based VHDL analyzer (Bratton, 87), producing the VIA file, in Appendix C. The Build package of the simulator read the VIA file, producing the C files, depicted in Appendix D. The Build package required 0.5 CPU seconds, and the the resulting C files occupied 538 bytes of disk space. The C files were compiled and linked with the kernel of the simulator (Simulate package) in 5.3 CPU seconds, producing a 22,528 byte executable file. The executable file was run in non-interactive mode taking 0.3 CPU seconds. The results just presented are summarized in Table 4-4.

Phase	CPU Time (sec)		File Size (Bytes)	
	UNIX Version	VMS Version	UNIX Version	VMS Version
Build	0.5	92.74	538	58,368
Compile	5.3	53.68	22,528	242,176
Sim	0.3	6.44	512	8,192

Table 4-4. Computer Resource Requirements

4.3.3.1.2. VMS-based Simulator

The same VHDL description was analyzed by the VMS-based simulator, producing the required intermediate (IVAN) files. The **mg** command was executed, requiring 92.72 CPU seconds, producing files requiring 58,368 bytes. The build command of the VMS-based simulator was exercised, requiring 53.68 CPU seconds, and requiring 242,176 more bytes of disk space to produce the executable simulator. The simulator was executed, requiring 6.44 CPU seconds, and 8,192 bytes to store data for report generation. The data for the VMS-based simulation is also summarized in Table 4-4. The waveforms produced by both simulators are displayed for comparison in Figure 4-1.

4.3.3.2. Ring Oscillator Description

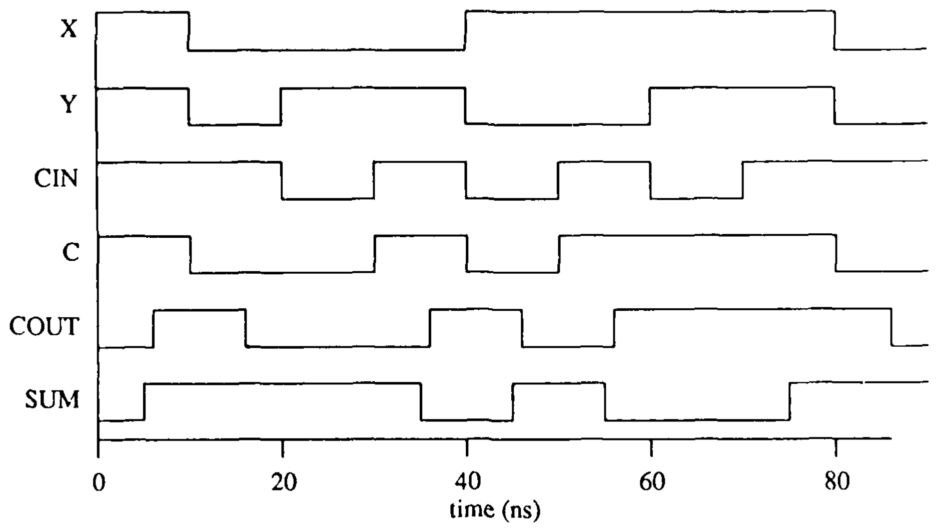
The VHDL description for the ring oscillator circuit evaluated is presented in Appendix E. The testing sequence for the full adder was repeated for the ring oscillator.

4.4. Performance Analysis

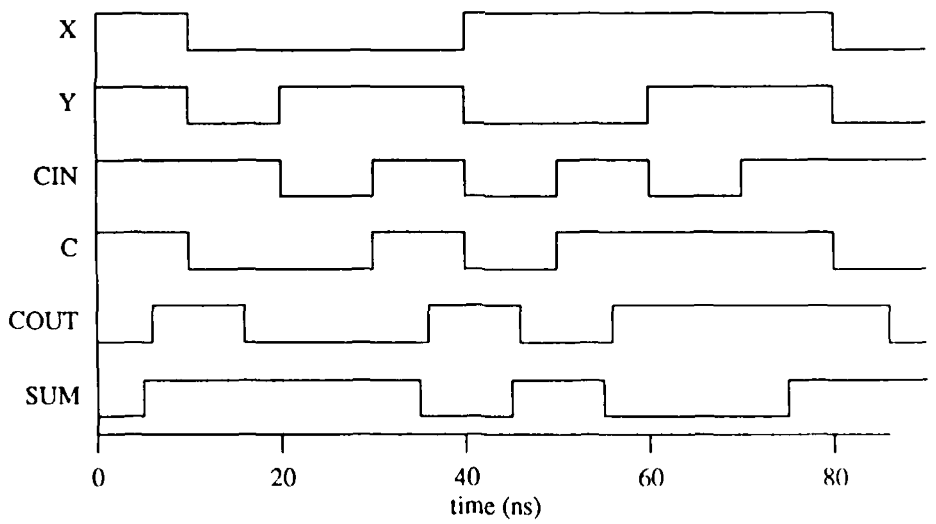
The performance of the UNIX-based simulator is evaluated in terms of the subset of the language implemented, the user's interface, and the CPU execution time and disk space requirements. Where appropriate, the UNIX-based simulator is compared to its VMS-based counterpart.

4.4.1. Language Features Implemented

As noted earlier, the UNIX-based simulator implements a subset of the language. While this may be viewed as a shortcoming, one must consider the need for a simulator capable of simulating 100% of the language. The UNIX-based simulator was designed primarily for the university community, where senior undergraduate and beginning graduate students would be introduced to the features and capabilities of VHDL. For this class of users, a simulator capable of simulating the major features of the language is sufficient. In a multi-user environment such as a university, qualities such as modest disk space requirements, short turn-around time, and ease of use are of primary importance. The features implemented in the UNIX-based simulator satisfy the needs of the user group it was designed for.

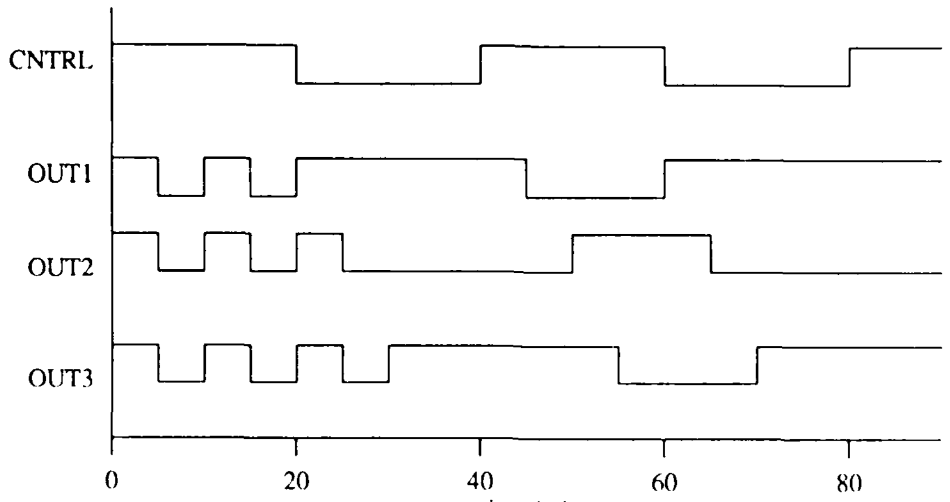


AFIT Simulator Output (VAX 11/785)
(Build - 0.5s CPU) (Compile - 5.3s CPU) (Sim - 0.3s CPU)

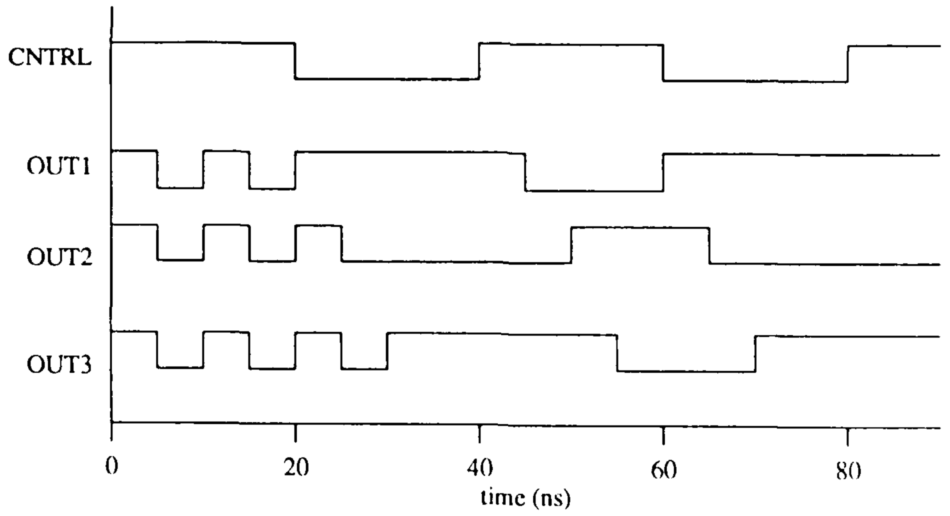


VMS System Simulator Output (VAX 11/785)
(Model Generate - 92.74s CPU) (Build - 53.68s CPU) (Sim - 6.44s CPU)

Figure 4-1. Full Adder Simulation Results



AFIT Simulator Output (VAX 11/785)
(Build - 0.6s CPU) (Compile - 5.3s CPU) (Sim - 0.3s CPU)



VMS System Simulator Output (VAX 11/785)
(Model Generate - 72.77s CPU) (Build - 55.84s CPU) (Sim - 5.93s CPU)

Figure 4-2. Ring Oscillator Results

4.4.2. User's Interface

The UNIX-based simulator provides the user with the option of specifying simulation parameters on the command line, as well as the ability to view intermediate results during a simulation. The user must include all simulation parameters, such as the simulation start and stop time, in the VHDL description. If the user desires to change any of these parameters, the VHDL description must be modified, analyzed, and simulated. Since the simulation start and stop time do not affect the syntax and semantics of the VHDL description, re-analysis is unnecessary. By providing the user with the opportunity to change these parameters interactively, the analyzer need not be run unnecessarily, and the simulator can be executed repeatedly for different test cases without modification.

Another attribute of the UNIX-based simulator that boosts productivity is the ability to provide a vector file of input stimuli to a circuit. With this feature, once the user has obtained a correctly analyzed VHDL description, it can simulated repeatedly, providing different sets of input stimuli via the vector file. Since the circuit does not change, analysis of the description and rebuilding the simulator is not performed. The VMS-based simulator provides input stimuli via signal assignments in the VHDL description. When the input stimuli change, the VHDL description must be changed and re-analyzed.

4.4.3. Utilization of Computer Resources

An examination of the results for the full adder and ring oscillator designs show the disparity in CPU time and disk storage requirements for the UNIX-based and VMS-based simulators. Minimizing these two parameters is particularly important in a multi-user environment such as a university where users are faced with resource and time constraints.

4.5. Summary

The UNIX-based simulator was evaluated based on the language features implemented, its user's interface, and its utilization of computer resources. Based on the results obtained, the UNIX-based simulator provides the university community users with a VHDL simulator capable of simulating a major subset of the language that is easy to use, is fast, and has modest storage requirements.

5. Conclusions and Recommendations

The purpose of this thesis was to develop a VHDL subset simulator to be incorporated into the growing AFIT VHDL Environment (AVE). The simulator supports common types, and a major subset of the sequential and concurrent statements of VHDL. The simulator is capable of reading the VHDL Intermediate Access (VIA) form produced by the VHDL analyzer, and producing C code which is compiled and linked with a simulator kernel, which is then executed, to produce a report file.

The completion of the VHDL simulator marks a significant contribution to the VHDL CAD tool set in the AVE. The simulator compares favorably to existing CAD tools.

5.1. Conclusions

The UNIX-based simulator successfully implements a subset of the VHDL language. The simulator was designed for runtime efficiency, efficient disk utilization, and ease of use. When a design was simulated by the UNIX-based version and the VMS-based version, the UNIX-based version was superior in all three areas, with both simulators producing the same results.

5.2. Recommendations

5.2.1. Implementation of the Complete VHDL Version 7.2

Since this thesis implements a subset of the language, it seems natural that the simulator be developed to simulate the entire language. This is the first major step that should be taken. The data structures used to develop the simulator were designed to later incorporate the features of the entire language.

5.2.2. Development of the Complete VHDL Analyzer

Without a complete analyzer, a complete simulator is not very useful. A complete analyzer should be developed before, or in parallel with the development of the complete simulator. If developed concurrently, the analyzer and simulator should maintain end-to-end throughput throughout their design. Features should be validated before others are added.

5.2.3. Efficiency

The simulator should be optimized for runtime efficiency. The data structures were designed runtime efficiency and workability in mind. At the higher levels of design, the data structures and algorithms were designed for runtime efficiency.

At the lower levels in the design, however, efficiency was occasionally sacrificed in order to produce a working product. For example, items such as search and sorting algorithms were not optimally implemented, but the software was designed so these modules could be easily redesigned and replaced.

5.2.4. IEEE Version

The implementation of this simulator supports the VHDL version 7.2. Future versions of the simulator should be implemented to adhere to the IEEE VHDL standard.

5.3. Summary

The development of the VHDL simulator makes a significant contribution to the AFIT VHDL Environment, and provide a baseline for the future development of the AVE. Additionally, the simulator, in conjunction with the analyzer, provide the university community and industry with a VHDL CAD tool set that operates under the UNIX operating system.

Appendix A - Simulator Test Suite

Included in Appendix A are the VHDL descriptions used to validate the simulator.

```
-- File := 1.v
-- This description tests the signal assignment statement and
--      process statement.
--

entity TEST1

   (X,Y:  in BIT;
    CIN:  in BIT;
    SUM:  out BIT;
    COUT: out BIT)

is
end TEST1;

architecture DATA_FLOW_IMPL of TEST1 is

   BLOCK_1:  block

      signal C: BIT;        -- Local signal declaration

   begin
     process (X, Y, CIN, C)
       begin
      SUM  <= X xor Y xor CIN after 5 ns;
      C    <= (Y and CIN) or (X and CIN) or (X and Y);
      COUT <= C after 6ns;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 2.v
-- This description tests the signal assignment statement.
--
entity TEST2

   (X,Y:  in BIT;
    CIN:  in BIT;
    SUM:  out BIT;
    COUT: out BIT)

is
end TEST2;

architecture DATA_FLOW_IMPL of TEST2 is

   BLOCK_1:  block

      signal C: BIT;       -- Local signal declaration

   begin

      SUM  <= X xor Y xor CIN after 5 ns;
      C    <= (Y and CIN);
      COUT <= C after 6ns;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 3.v
--  This file tests misc. features of VHDL, including
--      the process stmt, if stmt, enable, disable.

entity TEST3

  (X,Y:  in BIT;
   CIN:  in BIT;
   SUM:  out BIT;
   COUT: out BIT)

is
end TEST3;

architecture DATA_FLOW_IMPL of TEST3 is

   BLOCK_1:  block

      signal C: BIT;        -- Local signal declaration

   begin
    process (X, Y, C, CIN)
      begin
       if X = '1' then
       SUM  <= X xor Y xor CIN after 5 ns;
       elsif X = '0' then
       C    <= (Y and CIN);
       COUT <= C after 6ns;
       else
       C <= X;
       end if;
       enable X, Y;
       disable C;
     end process;
     process (X)
       begin
       if X = '0' then
         disable X;
       end if;
      end process;
    end block BLOCK_1;

 end DATA_FLOW_IMPL;
```

```
-- File := 4.v
--    This description tests the  enable, disable stmts.

entity TEST4

   (X,Y:  in BIT;
    CIN:  in BIT;
    SUM:  out BIT;
    COUT: out BIT)

is
end TEST4;

architecture DATA_FLOW_IMPL of TEST4 is

    BLOCK_1:  block

        signal C: BIT;        -- Local signal declaration

    begin
      process (X, Y, C)
        begin
       SUM  <= X after 5 ns;
       C    <= Y ;
       COUT <= C after 6 ns;
       enable X, Y;
         if C = '0' then
           enable C;
       elsif X = '1' then
           disable C;
       else
           enable X;
       end if;
      end process;
      process (X, Y)
        begin
          C <= transport '1' after 1 ns;
          C <= X;
        C <= Y after 2ns, X after 3ns;
      end process;

    end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 5.v
--  This description tests the case, enable, disable stmts.

entity TEST5

  (X,Y:  in BIT;
   CIN:  in BIT;
   SUM:  out BIT;
   COUT: out BIT)

is
end TEST5;

architecture DATA_FLOW_IMPL of TEST5 is

   BLOCK_1:  block

      signal C: BIT;        -- Local signal declaration

   begin
     process (X, Y, CIN, C)
       begin
      SUM  <= X xor Y xor CIN after 5 ns;
      C    <= (X and Y);
      COUT <= C after 6 ns;
      enable  X, Y;
      if X = '0' then
        disable X;
        end if;
      case Y is
        when '0'  => C <= '0' after 1 ns;
        when '1'  => C <= '1' after 3 ns;
        end case;
        if C = '0' then
         enable C;
      elsif Y = '1' then
         disable C;
      elsif X = '1' then
         enable X;
      end if;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 6.v
-- This description test the case stmt exhaustively.

entity TEST6

  (X,Y:  in BIT;
   CIN:  in BIT;
   SUM:  out BIT;
   COUT: out BIT)

is
end TEST6;

architecture DATA_FLOW_IMPL of TEST6 is

   BLOCK_1:  block

      signal C: BIT;        -- Local signal declaration
      signal D: INTEGER;

   begin
     process (C)
       begin
      C    <= (X and Y);
      case D is
        when 0 to 10   => C <= X;
        when abs(-11)  => C <= Y;
        when others    => enable C;
      end case;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 7.v
--  Three input XOR, testing process & signal assignment stmt.

entity XOR3

  (X,Y:  in BIT;                -- one-bit addends
   Z:  in BIT;                -- carry in
   F:  out BIT)               -- one-bit sum

is
end XOR3;

architecture DATA_FLOW_IMPL of XOR3 is

   BLOCK_1:  block

   begin
     process (X, Y, Z)
       begin
      F <= X xor Y xor Z after 41 ns;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
--  File := 8.v
--     This description tests the if statement.

entity IF_STMT

  (A,B:  in BIT;
   C : out BIT)

is
end IF_STMT;

architecture DATA_FLOW_IMPL of IF_STMT is

   BLOCK_1:  block

   begin
     process (A, B)
       begin
      if A = '0' then
        C <= A;
      elsif B = '1' then
        C <= '1';
      elsif A = '1' then
        C <= B;
      else
        C <= '0';
      end if;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 9.v
-- This description tests the signal assignment statement.
--
entity TEST1

is
end TEST1;

architecture DATA_FLOW_IMPL of TEST1 is

   BLOCK_1:  block

      signal A1,A2: BIT;        -- Local signal declaration
      signal B1, B2: integer;

   begin
     process (A1, A2, B1, B2)
       begin
      A1 <= A2 after 5ns, '0' after 10ns, A2 after 15ns;
      A2 <= transport A1 after 5ns;
      B1 <= B2 + 100 after 2ns;
      B2 <= B1;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
-- File := 10.v
-- To test the disable stmt.

entity TEST1

is
end TEST1;

architecture DATA_FLOW_IMPL of TEST1 is

   BLOCK_1:  block

      signal A1: BIT;        -- Local signal declaration

   begin
     process (A1)
       begin
      disable A1;
      A1 <= '1' after 3ns;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

```
--  File := 11.v
-- To test enable
--
entity TEST1

is
end TEST1;

architecture DATA_FLOW_IMPL of TEST1 is

   BLOCK_1:  block

      signal A1, A2: integer;        -- Local signal declaration

   begin
     process (A1, A2)
       begin
      A1 <= A1 + 1 after 2ns;
      A2 <= A2 + 2 after 1ns;
     end process;
     process (A1, A2)
       begin
       disable A1;
       if (A1 > 5) then
          enable A1;
       end if;
       A2 <= A2 + 2;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

Appendix B - Full Adder VHDL Description

```
entity Full_Adder

  (X,Y:  in BIT;
   CIN:  in BIT;
   SUM:  out BIT;
   COUT: out BIT)

is
end Full_Adder;

architecture DATA_FLOW_IMPL of Full_Adder is

   BLOCK_1:  block

      signal C: BIT;       -- Local signal declaration

   begin
     process (X, Y, CIN, C)
       begin
      SUM  <= X xor Y xor CIN after 5 ns;
      C    <= (Y and CIN) or (X and CIN) or (X and Y);
      COUT <= C after 6ns;
     end process;

   end block BLOCK_1;

end DATA_FLOW_IMPL;
```

Appendix C - Full Adder VIA File

```
0 34 48 3
0 __ANY__ 0 0 0 0 -1 -1 -1 -1 -1
1 STANDARD 3 4 0 1 -1 -1 -1 -1 -1
2 INTEGER 2 1 0 1 -2147483648 2147483647 2 -1 -1
3 CHARACTER 2 2 0 1 128 0 127 -1 -1
4 BIT 2 2 0 1 2 0 1 -1 -1
5 BOOLEAN 2 2 0 1 2 0 1 -1 -1
6 FALSE 1 3 0 1 5 0 -1 -1 7
7 TRUE 1 3 0 1 5 1 -1 -1 -1
8 SEVERITY_LEVEL 2 2 0 1 4 0 3 -1 -1
9 NOTE 1 3 0 1 8 0 -1 -1 10
10 WARNING 1 3 0 1 8 1 -1 -1 11
11 ERROR 1 3 0 1 8 2 -1 -1 12
12 FAILURE 1 3 0 1 8 3 -1 -1 -1
13 NS 1 3 0 1 14 1 -1 -1 15
14 TIME 2 5 0 1 0 2147483647 8 13 -1
15 US 1 3 0 1 14 1000 -1 -1 16
16 MS 1 3 0 1 14 1000000 -1 -1 17
17 S 1 3 0 1 14 1000000000 -1 -1 -1
18 0 1 3 0 1 4 0 -1 -1 19
19 1 1 3 0 1 4 1 -1 -1 -1
20 REAL 2 6 0 1 -2147483648 2147483647 20 -1 -1
21 NATURAL 2 1 0 1 0 2147483647 2 -1 -1
22 POSITIVE 2 1 0 1 1 2147483647 2 -1 -1
23 STRING 2 3 0 1 22 3 1 0 -1
24 BIT_VECTOR 2 3 0 1 21 4 0 0 -1
25 TEST1 3 1 0 25 30 -1 -1 -1 -1
26 Y 1 1 9 25 4 -1 -1 -1 27
27 X 1 1 9 25 4 -1 -1 -1 -1
28 CIN 1 1 9 25 4 -1 -1 -1 26
29 SUM 1 1 17 25 4 -1 -1 -1 28
30 COUT 1 1 17 25 4 -1 -1 -1 29
31 DATA_FLOW_IMPL 3 2 0 31 25 47 -1 -1 -1
32 BLOCK_1 1 4 0 32 47 -1 -1 -1 -1
33 C 1 1 0 33 4 -1 -1 -1 -1
0 0 1 -1 -1 -1 -1 -1
1 9 17 -1 5 7 -1 -1
2 9 20 -1 4 27 5 -1
3 9 20 -1 4 26 -1 -1
4 9 20 -1 4 28 3 -1
5 9 20 -1 4 33 4 -1
6 15 20 -1 -1 2 18 -1
7 9 22 -1 4 29 17 -1
8 9 22 -1 4 27 9 -1
9 9 22 -1 4 26 -1 -1
10 367 22 -1 4 8 11 -1
11 9 22 -1 4 28 -1 -1
12 367 22 -1 4 10 -1 -1
13 11 22 -1 14 -1 -1 5
14 291 22 -1 14 13 12 -1
```

```
15 13 22 -1 -1 14 -1 -1
16 21 22 -1 5 -1 -1 0
17 359 22 -1 -1 16 15 -1
18 4 22 -1 -1 7 36 -1
19 9 23 -1 4 33 35 -1
20 9 23 -1 4 26 21 -1
21 9 23 -1 4 28 -1 -1
22 294 23 -1 4 20 25 -1
23 9 23 -1 4 27 24 -1
24 9 23 -1 4 28 -1 -1
25 294 23 -1 4 23 -1 -1
26 339 23 -1 4 22 29 -1
27 9 23 -1 4 27 28 -1
28 9 23 -1 4 26 -1 -1
29 294 23 -1 4 27 -1 -1
30 339 23 -1 4 26 -1 -1
31 11 23 -1 14 -1 -1 0
32 291 23 -1 14 31 30 -1
33 13 23 -1 -1 32 -1 -1
34 21 23 -1 5 -1 -1 0
35 359 23 -1 -1 34 33 -1
36 4 23 -1 -1 19 44 -1
37 9 24 -1 4 30 43 -1
38 9 24 -1 4 33 -1 -1
39 11 24 -1 14 -1 -1 6
40 291 24 -1 14 39 38 -1
41 13 24 -1 -1 40 -1 -1
42 21 24 -1 5 -1 -1 0
43 359 24 -1 -1 42 41 -1
44 4 24 -1 -1 37 -1 -1
45 345 25 -1 -1 6 -1 -1
46 12 27 -1 -1 1 45 -1
47 301 27 32 -1 46 -1 -1
```

Appendix D - Full Adder C Files Generated

```
#include <stdio.h>
#include "sim_stru.h"
int *s27_pl;
int *s33_pl;
int *s28_pl;
int *s26_pl;
int s26_sens_proc[2];
int s27_sens_proc[2];
int s28_sens_proc[2];
int s29_sens_proc[1];
int s30_sens_proc[1];
int s33_sens_proc[2];
Driver *pdl[3];   int *psl[10];

sim_initialize(sig_array, pt)
Signal *sig_array[];
Process_table_item pt[];
{
  Signal *Newsig();
  Signal *s26;
  Signal *s27;
  Signal *s28;
  Signal *s29;
  Signal *s30;
  Signal *s33;
  char *strsave();
  extern int s26_sens_proc[];
  extern int s27_sens_proc[];
  extern int s28_sens_proc[];
  extern int s29_sens_proc[];
  extern int s30_sens_proc[];
  extern int s33_sens_proc[];

  int i;
  Driver *Newdrv();
  int p45();
  extern Driver *pdl[]; extern int *psl[];
  Driver *d29;
  Driver *d30;
  Driver *d33;
 /* end of declarations */

 /* create signals */
  s26 = Newsig(); s26->signame = strsave("Y");
  s27 = Newsig(); s27->signame = strsave("X");
  s28 = Newsig(); s28->signame = strsave("CIN");
  s29 = Newsig(); s29->signame = strsave("SUM");
  s30 = Newsig(); s30->signame = strsave("COUT");
  s33 = Newsig(); s33->signame = strsave("C");
  sig_array[0] = s33;
  sig_array[1] = s28;
  sig_array[2] = s30;
  sig_array[3] = s29;
  sig_array[4] = s27;
  sig_array[5] = s26;
/* declaring drivers */
  d29 = Newdrv(); d29->sig_ptr = s29;     s29->drv_ptr = d29;
  d30 = Newdrv(); d30->sig_ptr = s30;     s30->drv_ptr = d30;
  d33 = Newdrv(); d33->sig_ptr = s33;     s33->drv_ptr = d33;
/* sim02.c */
  s26_sens_proc[0] = 0;
  s27_sens_proc[0] = 0;
  s28_sens_proc[0] = 0;
  s29_sens_proc[0] = 0;
  s30_sens_proc[0] = 0;
  s33_sens_proc[0] = 0;
  s26->sens_proc = s26_sens_proc;
  s27->sens_proc = s27_sens_proc;
  s28->sens_proc = s28_sens_proc;
  s29->sens_proc = s29_sens_proc;
  s30->sens_proc = s30_sens_proc;
  s33->sens_proc = s33_sens_proc;
  pt[1].proc_ptr = p45;
/* found a sens list for a process */
  s27_sens_proc[++s27_sens_proc[0]] = 1;
      s27_p1 = &s27_sens_proc[s27_sens_proc[0]];
  s33_sens_proc[++s33_sens_proc[0]] = 1;
      s33_p1 = &s33_sens_proc[s33_sens_proc[0]];
  s28_sens_proc[++s28_sens_proc[0]] = 1;
      s28_p1 = &s28_sens_proc[s28_sens_proc[0]];
  s26_sens_proc[++s26_sens_proc[0]] = 1;
      s26_p1 = &s26_sens_proc[s26_sens_proc[0]];
  pd1[0] = d29;
  ps1[0] = s27->cur_val;
  ps1[1] = s26->cur_val;
  ps1[2] = s28->cur_val;
  pd1[1] = d33;
  ps1[3] = s26->cur_val;
  ps1[4] = s28->cur_val;
  ps1[5] = s27->cur_val;
  ps1[6] = s28->cur_val;
  ps1[7] = s27->cur_val;
  ps1[8] = s26->cur_val;
  pd1[2] = d30;
  ps1[9] = s33->cur_val;
/* process table filling of driver & signal list */
  pt[0].next = &pt[0];
  for(i=1; i<2; i++)
    pt[i].next = NULL;
  pt[1].dlist = pd1;    pt[1].slist = ps1;
}

p45(dlist,slist)
Driver *dlist[];
```

```
int *slist[];
{
  Transact *Newtrans();
  extern TIME simtime;
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1->future_time = simtime + 5;
    *newtrans1->val = xor(xor(*slist[0], *slist[1]), *slist[2]);
    post_trans(dlist[0], newtrans1,transport);
  }
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1->future_time = simtime + 0;
    *newtrans1->val = or(or(and(*slist[3], *slist[4]), and(*slist[5],
      *slist[6])), and(*slist[7], *slist[8]));
    post_trans(dlist[1], newtrans1,transport);
  }
  {
    BOOLEAN transport = FALSE;
    Transact *newtrans1; /*wave*/
    newtrans1 = Newtrans();
    newtrans1->future_time = simtime + 6;
    *newtrans1->val = *slist[9];
    post_trans(dlist[2], newtrans1,transport);
  }
}
```

Appendix E - Ring Oscillator VHDL Description

```
entity Ring_Oscillator is
end Ring_Oscillator;

architecture df_impl of Ring_Oscillator is
   l:block
      signal st_enable, bn_enable : bit;

        signal CNTRL, Out1, Out2, Out3 : Bit;

begin

        CNTRL  <= transport '0' after 30 ns,
                            '1' after 50 ns,
                            '0' after 70 ns,
                            '1' after 90 ns;

            Out1   <= CNTRL nand Out3 after 5 ns;
            Out2   <= not Out1 after 5 ns;
            Out3   <= not Out2 after 5 ns;

    end block l;
end df_impl;
```

References

Aylor J.H. *et al.* "VHDL -- Feature Description and Analysis" *IEEE Design and Test of Computers,* 3: 17-27 (April 1986).

Barbacci Mario R. and Takao Uehara. "Computer Hardware Description Languages: The Bridge Between Software and Hardware" *IEEE Computer,* 10: 6-8 (June 1977).

Booch, Grady. *Software Engineering with Ada, 2ed.* Melno Park, CA. Benjamin Cummins Publishing Co., 1987.

Bratton, Capt Randolph M. *A Production-Quality UNIX Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) Subset Analyzer.* MS Thesis AFIT/GCS/MA/87D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987.

Carter, Lt Col Harold W. *et al. 1986 Research Report AFIT VHDL/DB/DBMS Research.* AFIT-ENC-TR-87-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, January 1987.

Chu, Yaohan. "Why Do We Need Computer Hardware Description Languages?" *IEEE Computer,* 7: 18-22 (December 1974).

d'Abreu, Michael. "Gate-Level Simulation," *IEEE Design And Test,* 2: 63-71 (December 1985).

Dewey, Allen and Anthony Gadient. "VHDL Motivation" *IEEE Design and Test of Computers,* 3: 12-16 (April 1986).

Frauenfelder, Capt Deborah J. *VHDL Language Analyzer.* MS Thesis AFIT/GCS/ENG/86D-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

Gilman, Alfred S. "VHDL -- The Designer Environment" *IEEE Design and Test of Computers,* 3: 42-47 (April 1986).

Intermetrics, Inc. *Simulator Program Specification.* U.S. Air Force Contract F33615-83-C-1003. Bethesda, MD., 30 July 1984.

Intermetrics, Inc. *VHDL Language Reference Manual - Revised Version 7.2.* U.S. Air Force Contract IR-MD-045-3. Bethesda, MD., 1 August 1985.

Intermetrics, Inc. *VHDL User's Manual: Volume 1 - Tutorial.* U.S. Air Force Contract F33615-83-C-1003. Bethesda, MD., 1 August 1985.

Intermetrics, Inc. *VHDL User's Manual: Volume 2 - User's Reference Guide.* U.S. Air Force Contract F33615-83-C-1003. Bethesda, MD., 1 August 1985.

Kernighan, B.W. and Ritchie, D.M. *The C Programming Language.* Englewood Cliffs, New Jersey. Prentice-Hall, Inc., 1978.

Lipovski, G.J. "Hardware Description Languages: Voices from the Tower of Babel" *IEEE Computer,* 10: 14-17 (June 1977).

Lynch, Maj William Leo. *VHDL Prototype Simulator.* MS Thesis AFIT/GCS/ENG/86D-15. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach, 2ed.* New York. McGraw-Hill Book Co., 1987.

Shahdad, Moe *et al.* "VHSIC Hardware Description Language" *IEEE Computer*, 18: 94-103 (February 1985).

Su, Stephen Y.H. "A Survey of Computer Hardware Description Languages in the U.S.A." *IEEE Computer*, 7: 45-49 (December 1974).

Su, Stephen Y.H. "Hardware Description Language Applications: An Introduction and Prognosis" *IEEE Computer*, 10: 10-13 (June 1977).

Vita

Harvey H. Kodama was born on July 6, 1963 in Honolulu, Hawaii. He graduated from Iolani School, Honolulu in 1981 and received a Bachelor of Science in Electrical Engineering degree from University of Southern California, Los Angeles in May 1986. Upon graduation, he was commissioned a second lieutenant in the U.S. Air Force through the ROTC program, and was subsequently assigned to Air Force Institute of Technology, Wright-Patterson AFB, Ohio.

Permanent Address:

1438 Kaumoli St.
Pearl City, HI 96782

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/87D-33	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, OH 45433	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING / SPONSORING ORGANIZATION AF Wright Aeronautical Labs	8b. OFFICE SYMBOL *(If applicable)* AFWAL/AADE-3
9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB, OH 45433	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE *(Include Security Classification)*
A UNIX-BASED INTERACTIVE VHDL SIMULATOR

12 PERSONAL AUTHOR(S)
Kodama, Harvey H., 2Lt, USAF

13a. TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
M.S. Thesis	FROM Mar 87 TO Dec 87	1987 Dec 7	76

16 SUPPLEMENTARY NOTATION This thesis is submitted in partial fullfillment of the requirements for the Master of Science degree at Air Force Institute of Technology, Wright Patterson, AFB, OH.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
VHDL
Simulator Simulation Model HARDWARE Desc. Lang.

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Advisor: Joseph W. DeGroat, Major, USAF

[illegible] 24 Feb 88

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Joseph W. DeGroat, Maj, USAF	22b TELEPHONE *(Include Area Code)* (513) 255-6913
22c OFFICE SYMBOL AFIT/ENG	

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

This thesis effort investigated and implemented a UNIX-based VHDL subset interactive simulator. The simulator was written in the C programming language. The simulator is capable of handling many of the VHDL operators, sequential assignment statements, and concurrent statements. Although the simulator is a subset, the data structures were designed to incorporate the features of the complete VHDL version 7.2.